informatique(s)

MP - MP* / 2014 - 2015

Mis à jour le 18 décembre 2014 à 18:24

Algo: Niveau II

Guillaume CONNAN - Lycée Clemenceau

UNIVERSITÉ DE NANTES

TABLE DES MATIÈRES

THÈME

1

Structures de données linéaires

En 1920, le mathématicien-logicien-philosophe polonais Jan Łukasiewicz (1878-1956) invente la notation préfixée alors qu'il est ministre de l'éducation (on a le droit de rêver...).
35 ans plus tard, le philosophe et informaticien (!) australien (!!!) Charles Hamblin (1922 - 1985) s'en inspire.
Il a en effet en sa possession l'un des deux ordinateurs présents à l'époque en Australie. Il se rend compte que la saisie de calculs avec les opérateurs en notation infixée induit de nombreuses erreurs de saisie par oubli de parenthèses. Il pense aussi à la (très petite) mémoire des ordinateurs de l'époque.
Il trouve alors son inspiration dans le travail de Łukasiewicz pour éviter les parenthèses et il pense à mettre les opérateurs en position préfixée : ainsi il introduit la notion de *pile* (ou *stack* ou Lifo) qui économise le nombre d'adresses mémoire nécessaires.
C'est la naissance de la Notation Polonaise Inversée (NPI ou RPN). Elle économise également la taille des composants électroniques des portes logiques.
Son seul inconvénient : les mauvaises habitudes prises d'utiliser des notations infixées...
Un autre grand avantage : il faut comprendre le calcul avant de l'exécuter :-)

1 Tableaux

Jusqu'à maintenant, nous avions à disposition une seule structure de données : le tableau.
En Python, les tableaux s'appellent malheureusement des listes qu'il ne faut pas confondre avec la structure appelée habituellement liste en informatique !(Ah....Python...)
Commentez les résultats qui suivent :

Python
```
def m0():
    return sum([i + 3 for i in range(1000)])

def m1():
    return np.sum(np.arange(1000) + 3)

def m2():
    return np.sum(map(lambda x: x + 3,range(1000)))
```

Python
```
In [142]: %timeit m0()
10000 loops, best of 3: 63 µs per loop

In [143]: %timeit m1()
100000 loops, best of 3: 17 µs per loop

In [144]: %timeit m2()
100000 loops, best of 3: 12.8 µs per loop
```

2 Création de listes linéaires non bornées

2 1 Version 1

On voudrait construire un objet qui ressemble à un tableau mais dont on n'aurait pas à donner la taille a priori. Il doit ressembler à un ensemble d'éléments de *même type* placés dans un *ordre donné*.
Par exemple, on sait calculer la moyenne de trois notes mais comment faire pour calculer la moyenne d'un nombre quelconque de notes ?
On pourrait donner la définition mathématique suivante :

Définition 1 - 1

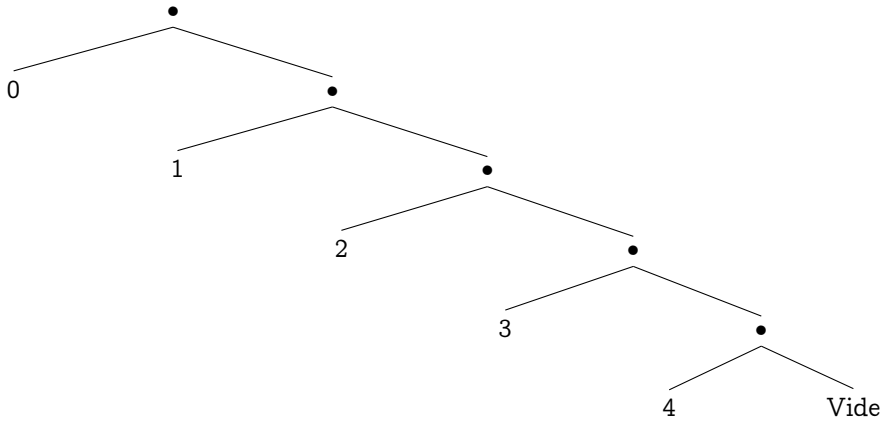
Liste linéaire
Une liste linéaire sur un ensemble E est une suite finie $x_0, x_1, ..., x_n$ d'éléments de E.

Cela n'est cependant pas suffisant pour manipuler ce genre d'objet.
On pourrait aussi donner cette définition :

Définition 1 - 2

Liste linéaire : version récursive
Une liste linéaire sur un ensemble E est soit le *Vide* soit un élémentde E suivi d'une liste.

(*Comparez avec cette définition d'un mot : c'est le mot vide ou une lettre suivie d'un mot...*).
Cette notion de liste est si importante en informatique qu'un des langages les plus marquants de son histoire s'appelle LISP (*LISt Processor* cf Brygoo et coll. [2004]).
Nous allons en donner une *spécification* beaucoup plus riche.
D'abord, on peut se représenter nos listes de manière plus précise. La liste $(0, 1, 2, 3, 4)$ ressemble en fait à ça :



Il nous faut disposer d'une liste vide. Il faut pouvoir ajouter un élément en tête, sélectionner la tête, sélectionner la liste privée de la tête, tester si une liste est vide, calculer sa longueur.
On ne pourra pas demander la tête ou la queue d'une liste vide.
La longueur de la liste vide doit être nulle. Une liste ayant une tête n'est pas vide. La longueur d'une liste à laquelle on a ajouté un élément en tête est incrémentée d'une unité.
Si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale.
On peut résumer ces spécifications dans un langage un peu plus formel (cf Dufourd et coll. [1995]).

— **Sorte :**
  Liste (liste d'objets de sorte E)
— **Opérations :**
  Vide : : Liste
  ajoute : : Liste E -> Liste
  tête : : Liste -> E
  queue : : Liste -> Liste
  estVide : : Liste -> Bool
  taille : : Liste -> Nat
— **Préconditions :**
  pré tête(liste) = ¬ estVide(liste)
  pré queue(liste) = ¬ estVide(liste)
— **Axiomes :**
  estVide(Vide) = Vrai
  taille(Vide) = 0
  estVide(ajoute(x,l)) = Faux
  taille(ajoute(x,l)) = taille(l) + 1
  tête(ajoute(x,l)) = x
  queue(ajoute(x,l)) = l

Il ne reste plus qu'à traduire cela dans un langage de programmation.
Malheureusement, il va falloir le faire en Python ;-)
On peut construire une liste comme un couple en utilisant les n-uplets de Python (ou *tuple*).

Python

```
#! python3.4
#-*- coding: utf-8 -*-

##############################################################
#####
#####          IMPLÉMENTATION SPÉCIFIQUE (ICI AVEC TUPLES)
#####
##############################################################

vide = None
```

```
# testeur

def est_vide(lst):
    """
    teste si une liste est vide
    """
    return lst == vide

# sélecteurs

def tete(lst):
    """
    renvoie le premier élément d'une liste (sa tête)

    >>> tete((1,(2,(3,vide))))
    1
    """
    if est_vide(lst):
        raise ValueError("La liste vide n'a pas de tête")
    else:
        return lst[0]

def queue(lst):
    """
    renvoie la liste privée de sa tête : c'est encore une liste

    >>> queue((1,(2,(3,vide))))
    (2, (3, vide))
    """
    if est_vide(lst):
        raise ValueError("La liste vide n'a pas de queue")
    else:
        return lst[1]

# insertion

def insere(el,lst):
    """
    insere un élement en tête de liste

    >>> insere(12, (1,(2,(3,vide))))
    (12, (1, (2, (3, vide))))
    """
    return (el,lst)

##########################################################################
####
#### BARRIÈRE D'ABSTRACTION : NE DÉPEND PAS DE L'IMPLÉMENTATION DES LISTES
####
##########################################################################

def concat(lst1,lst2):
    """
    concatène deux listes

    >>> concat((0,(1,(2,(3,(4,None))))) ,(10,(11,(12,(13,(14,None))))) )
    (0, (1, (2, (3, (4, (10, (11, (12, (13, (14, None))))))))))
    """
    if est_vide(lst1):
        return lst2
    else:
        return insere(tete(lst1), concat(queue(lst1),lst2))

### Petits sucres pour faciliter l'écriture et la lecture

def print_liste(li):
```

```
    """
    joli affichage d'une liste :

    >>> print_liste(liste(1,liste(2,liste(3,vide))))
    '< 1 2 3 >'
    """
    lst = li
    c = '< '
    while lst != vide:
        c += repr(tete(lst)) + ' '
        lst = queue(lst)
    return c[:-1] + ' >'

def from_listp(lst_p):
    """
    Construit une liste à partir d'un iterable python

    >>> l = from_iter(range(10))
    >>> print_liste(l)
    '< 0 1 2 3 4 5 6 7 8 9 >'
    """
    l = vide
    for k in lst_p[::-1]:
        l = insere(k,l)
    return l
```

Remarque

Dans ce cas, pour être plus efficace **sur Python** (on divise par 3 le nombre d'appels), on peut éviter de multiplier les appels aux fonctions intermédiaires :

Python

```
def applique(lst,f):
    if not lst:
        return None
    else:
        return (f(lst[0]),applique(lst[1],f))
```

MAIS on perd un peu le sens de la construction des listes qui est notre propos et surtout la *barrière d'abstraction* qui nous permettra de conserver les mêmes fonctions quelque soit notre version d'implémentation des listes.

Certaines fonctions sont commentées car elles fonctionnent toutes de la même manière. Nous allons les unifier dans la Recherche 1 - 5 page 22.

2 2 Version 2

Voici une deuxième méthode qui construit une liste chaînée sous forme d'une fonction :

Python

```
def cons(a, b):
    """
    construit une fonction qui renvoie a si son argument vaut '1' et b sinon
    cons :: (type1, type2) -> string -> type1
                           -> type3 -> type2
    C'est une manière de construire une sorte de tuple

    >>> cons('bon','jour')('1')
    'bon'
    >>> cons(23,45)('1')
    23
    >>> cons(23,45)('Hasta la vista baby')
    45
    """
    return lambda x : a if x == '1' else b
```

```
def prem(consab):
    """
    construit la fonction qui retourne le premier élément d'une paire fonctionnelle

    >>> prem(cons('bon','jour'))
    'bon'
    """
    return consab('1')

def deuz(consab):
    """
    construit la fonction qui retourne le 2e élément d'une paire fonctionnelle

    >>> deuz(cons('bon','jour'))
    'jour'
    """
    return consab('2')

##########################################################
#####
##### IMPLÉMENTATION SPÉCIFIQUE (ICI AVEC LA FONCTION CONS)
#####
##########################################################


vide = lambda : "n'importe quoi"
"""vide est une fonction sans argument"""

def liste(element, list):
    """
    Une liste est une FONCTION construite à partir de cons

    >>> l = liste(1,liste(2,liste(3,vide)))
    """
    return cons(element, list)

def est_vide(lst):
    return lst == vide

def tete(lst):
    if est_vide(lst):
        raise ValueError("La liste vide n'a pas de tête")
    else:
        return prem(lst)

def queue(lst):
    if est_vide(lst):
        raise ValueError("La liste vide n'a pas de queue")
    else:
        return deuz(lst)

def insere(el,lst):
    return liste(el,lst)
```

2 3 Version 3

Je vous livre une autre implémentation qui va nous initier à la programmation objet :

Python

```
class Liste(object):
    def __init__(self, tete , queue = Vide):
        """
        Une liste est définie par:
        - son premier élément
        - la liste des autres éléments.
```

```
        In [1]: l = Liste(1,Liste(2,Liste(3,Vide)))
        La liste doit avoir des éléments de même type
        """
        self.__tete  = tete
        self.__queue = queue
        if (self.__tete is None):
            raise SyntaxError("la liste doit avoir la forme Liste(el, Liste) ou être
                Vide")
        if not ((self is Vide)
                or (queue is Vide)
                or isinstance(tete, type(queue.__tete))):
            raise TypeError("La liste doit être de type homogène")

    def estVide(self):
        return (self is Vide)

    def queue(self):
        return self.__queue

    def tete(self):
        return self.__tete

    def insere(self,x):
        return Liste(x,self)

    def __str__(self):
        lst = self
        c = '< '
        while lst != Vide:
            c += repr(lst.__tete) + ' '
            lst = lst.__queue
        return c[:-1] + ' >'
```

Il faut également traiter le cas vide :

Python

```
class Vide(object):
    def __str__():
        return ""
    def tete():
        raise TypeError("La liste vide n'a pas de tête")
    def queue():
        raise TypeError("La liste vide n'a pas de queue")
    def insere(x):
        return Liste(x,Vide)
    def estVide():
        return True
```

Ouch ! Il y a du boulot...
Pour les heureux inscrits en option Info qui utilisent Ocaml, on peut comparer avec ce que cela pourrait donner dans ce beau langage :

OCaml

```
type 'a liste =
  |Vide
  |L of ('a * 'a liste);;

let ajoute el els = L(el,els);;

exception ListeVide;;

let tete = function
  |Vide -> raise ListeVide
  |L(el,els) -> el;;
```

```
let queue = function
  |Vide -> raise ListeVide
  |L(el,els) -> els;;

let rec taille = function
  |Vide -> 0
  |L(el,els) -> 1 + (taille l);;
```

et c'est tout. Par exemple :

OCaml

```
utop[11]> let l = L(1,Vide);;
val l : int liste = L (1, Vide)
utop[12]> let l1 = ajoute 2 l;;
val l1 : int liste = L (2, L (1, Vide))
utop[13]> taille l1;;
- : int = 2
utop[14]> tete Vide;;
Exception: ListeVide.
```

Mais revenons à notre serpent... Il cache dans son ventre des notions qui vont être abordées cette année. Par exemple :

— un style de programmation orienté objet ;

— de la récursion ;

— une étude de la complexité des fonctions que l'on va créer.

(*Les commentaires du prof au tableau viennent ici éclairer l'étudiant perplexe.*)

2 4 Version 4

Python

```
from copy import copy

class Noeud(object):
    """Un noeud a une valeur et pointe vers un autre noeud"""

    def __init__(self,elt = None, sui = None):
        self.__valeur  = elt
        self.__suivant = sui

    def get_valeur(self):
        """accesseur"""
        return self.__valeur

    def set_valeur(self,v):
        """mutateur"""
        self.__valeur = v

    def get_suivant(self):
        return self.__suivant

    def set_suivant(self,s):
        self.__suivant = s

    def  __str__(self):
        """ on affiche seulement la valeur du noeud """
        return str(self.get_valeur())

class Liste(object):

    def __init__(self):
        self.__tete = None

    def est_vide(self):
```

```
        return self.__tete is None

    def insere(self,elt):
        """La tête a pour valeur la valeur entrée et pointe vers l'ancienne tête"""
        n = Noeud(elt)
        n.set_suivant(self.__tete)
        self.__tete = n

    def val_tete(self):
        return self.__tete.get_valeur()

    def decapite(self):
        """On enlève la tête de la liste"""
        if self.__tete is None:
            raise TypeError("La liste vide n'a pas de queue")
        else:
            t = self.__tete
            self.__tete = t.get_suivant()
            t = None

    def queue(self):
        """on renvoie la queue sans modifier la liste"""
        l = copy(self)
        l.decapite()
        return l

    def __str__(self):
        s = ""
        t = self.__tete
        while not (t is None):
            s = s + str(t) + (', ' if t.get_suivant() is not None else '')
            t = t.get_suivant()
        return "<" + s + ">"
```

3 NPI

3 1 Arbre de calcul

Le rôle des parenthèses est le plus souvent primordial sur machine :

Python

```
In [54]: 5 + 4 * 3
Out[54]: 17
```

et ce quelque soit le langage car des règles de priorité ont été établies.

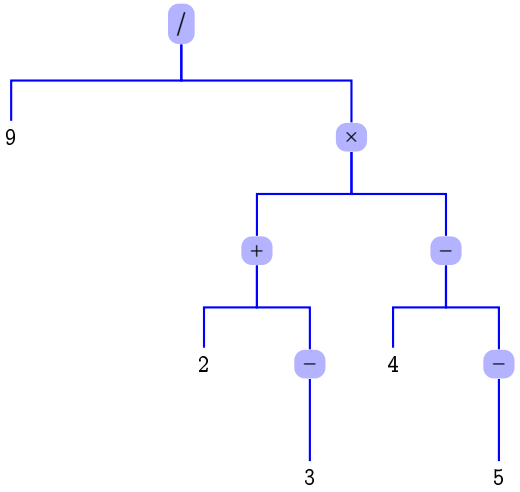
Sinon, des parenthèses auraient dû être mises :

Python

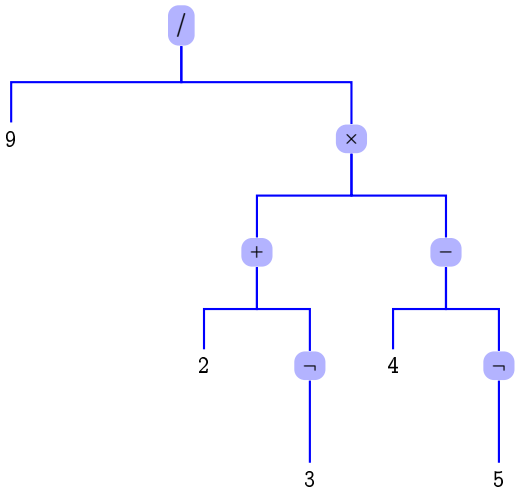
```
In [55]: 9 / 2 + (-3) * 4 - (-5)
Out[55]: -2.5

In [56]: 9 / ((2 + (-3)) * (4 - (-5)))
Out[56]: -1.0
```

Or dans un calcul plus compliqué, ne pas se soucier des parenthèses peut entraîner de mauvaises surprises.

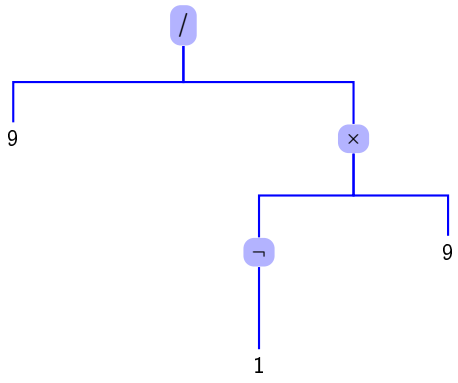
Il est sûrement plus clair de visualiser le calcul à l'aide d'un arbre



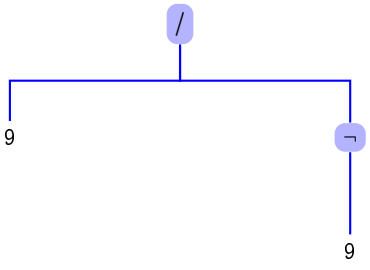
L'usage du − est ici ambigu : cet opérateur est parfois binaire, parfois unaire. Pour éviter ce problème, on peut par exemple utiliser des symboles différents :



On réduit alors l'arbre en remontant des feuilles vers la racine :



puis



et finalement ¬1.
À partir de l'arbre, on obtient donc une *séquence* de lecture *infixée* en *profondeur* :

$$9/((2+\neg 3)\times(4-\neg 5))$$

L'emploi des parenthèses agit comme une sorte de *retardateur* de l'évaluation.
Comment une machine évalue une telle expression ? C'est en fait assez compliqué car il faut disposer de règles de priorité, d'associativité, d'une grammaire qui procède donc *récursivement*...
Un autre moyen est de convertir cette expression en notation post-fixée qui utilise une *pile*.

3 2 Pile : découverte

Une pile (en anglais *stack*) est une structure de données permettant l'insertion d'un élément et la suppression du dernier élément inséré.
L'acronyme anglais associé est LIFO (*Last In First Out*).
Alan TURING proposa dès 1945 les termes *bury* et *unbury* pour désigner les deux méthodes basiques de traitement des piles.
On utilise en français *empiler* et *dépiler* alors que les termes anglais actuellement utilisés sont *push* et *pop*.
Imaginez une partie de cartes qui nécessite une pioche. La seule carte qui nous intéresse est celle du dessus. Piocher une carte consiste à retirer la carte du dessus et à la lire (dépiler) ou rajouter une carte *sur* le tas (empiler). Toute information supplémentaire est superflue.
De nombreuses situations sont similaires en informatique et.... *dans la vraie vie*.
D.E. KNUTH propose cette illustration (page 240 de Knuth [1997])

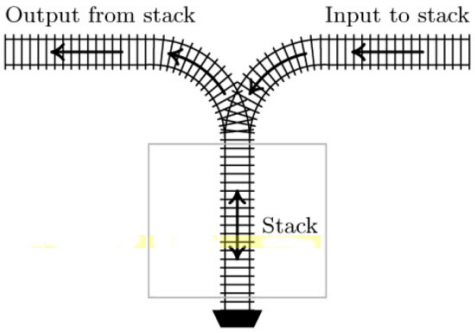
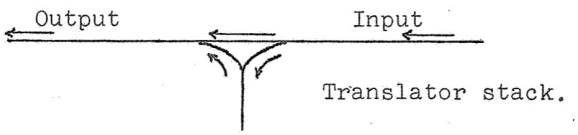


Fig. 1. A stack represented as a railway switching network.

qui reprend l'article fondateur de E.W. *Dijkstra* (Dijkstra [1961]) dont nous reparlerons bientôt :

The translation process shows much resemblance to shunting at a three way railroad junction of the following form



Mais bon, on utilise le terme *pile* en référence à un pile d'assiette : la première que l'on saisira sera la dernière à avoir été empilée.

3 3 Pile et NPI

Reprenons notre expression $9/((2+\neg 3)\times(4-\neg 5))$.
La plupart des compilateurs vont plutôt la lire ainsi : **9 2 3 NEG + 4 5 NEG - * /** en utilisant une pile.
Il faut juste connaître l'*arité* de chaque opérateur, i.e. leur nombre d'opérandes, qui doit être fixe (c'est pourquoi on doit absolument distinguer **-** de **NEG** et éviter l'habituelle *surcharge* de l'opérateur **-**).

1. On lit l'expression de gauche à droite ;
2. On empile les opérandes ;
3. Dès qu'on lit un opérateur, on l'applique aux opérandes présents sur la pile selon son arité ;
4. On s'arrête quand on n'a plus rien à lire ;
5. on renvoie le dernier élément présent dans la pile.

Voici par exemple les différentes étapes du calcul effectué à l'aide du *calculator* d'Emacs :

1. On entre **9** , **2** puis **3** :

Emacs
```
Emacs Calculator Mode
3:  9
2:  2
1:  3
.
```

2. **NEG**

Emacs
```
Emacs Calculator Mode
3:  9
2:  2
1:  -3
.
```

3. **+**

Emacs
```
Emacs Calculator Mode
2:  9
1:  -1
.
.
```

4. **4** puis Entrée

Emacs
```
Emacs Calculator Mode
3:  9
2:  -1
1:  4
.
```

5. **5** puis Entrée

Emacs
```
Emacs Calculator Mode
4:  9
3:  -1
2:  4
1:  5
```

6. **NEG**

Emacs
```
Emacs Calculator Mode
4:  9
3:  -1
2:  4
1:  -5
```

7. **-**

Emacs
```
Emacs Calculator Mode
3:  9
2:  -1
1:  9
.
```

8. *****

Emacs
```
Emacs Calculator Mode
2:  9
1:  -9
.
.
```

9. **/**

Emacs
```
Emacs Calculator Mode
1: -1
.
.
.
```

Plusieurs avantages :

— pas besoin de parenthèses, de règles d'associativité, de priorité ;

— on voit au sommet de la pile les résultats intermédiaires ;

— le calcul est directement implémentable sur machine.

Au collège, $9/((2+\neg 3)\times(4-\neg 5))$ serait lu :

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

ce qui peut se coder en :

`/ 9 * + 2 NEG 3 - 4 NEG 5`

C'est la méthode introduite en 1929 par Jan Łukasiewicz. On a dénommé *notation polonaise* cette méthode d'évaluation d'une expression en son honneur. On utilise également la dénomination *notation pré-fixée*. Cela explique aussi l'appellation *notation polonaise inversée* de la notation post-fixée.
Ici aussi, les opérateurs doivent être d'arité fixe et l'évaluation se fait à l'aide d'une pile.

Recherche

Décomposez le calcul **`/ 9 * + 2 NEG 3 - 4 NEG 5`**. Comment l'organiser sur machine ? Comparez avec la NPI. Expliquez alors le choix fait par Charles Hamblin d'utiliser plutôt la NPI sur machine.

3 4 Une première classe Pile

Nous allons utiliser les méthodes `pop` et `append` qui existent dans la classe `list` de Python. Cela sera notre première découverte des classes d'objet en Python.

Python
```
class Pile(list):

    def est_vide(self):
        return self == []
```

```
    def empile(self,elmt):
        return self.append(elmt)

    def depile(self):
        return self.pop()
```

RECHERCHES

Recherche 1 - 1 Adresse

Commentez ! Quelles précautions prendre ?

Python

```
In [22]: t = [0,1,2,3]

In [23]: id(t)
Out[23]: 140162835709256

In [24]: tt = t

In [25]: id(tt)
Out[25]: 140162835709256

In [26]: t += [4]

In [27]: t
Out[27]: [0, 1, 2, 3, 4]

In [28]: id(t)
Out[28]: 140162835709256

In [29]: tt
Out[29]: [0, 1, 2, 3, 4]

In [30]: t = t + [5]

In [31]: t
Out[31]: [0, 1, 2, 3, 4, 5]
```

```
In [32]: tt
Out[32]: [0, 1, 2, 3, 4]

In [33]: id(t)
Out[33]: 140162734366856

In [34]: id(tt)
Out[34]: 140162835709256

In [35]: import copy as cp

In [39]: ttt = cp.copy(t)

In [40]: id (ttt)
Out[40]: 140162835291592

In [41]: t += [6]

In [42]: t
Out[42]: [0, 1, 2, 3, 4, 5, 6]

In [43]: tt
Out[43]: [0, 1, 2, 3, 4]

In [44]: ttt
Out[44]: [0, 1, 2, 3, 4, 5]
```

Recherche 1 - 2 Performances

Commentez :

Python

```
import numpy as np

def v1():
    l = [0,1,2]
    l += [3]
def v2():
    l = [0,1,2,None]
    l[3] = 3
def v3():
    l = [0,1,2]
    l.append(3)
def v4():
    l1 = np.array([0,1,2])
    l2 = np.array([3])
    np.concatenate((l1,l2))
def v5():
    l = np.array([0,1,2,None])
    l[3] = 2
def v6():
    l = np.arange(4)
    l[3] = 12
def v7():
    l = it.chain([0,1,2],[3])
```

Python

```
In [45]: %timeit v1()
1000000 loops, best of 3: 320 ns per loop

In [46]: %timeit v2()
10000000 loops, best of 3: 179 ns per loop

In [47]: %timeit v3()
1000000 loops, best of 3: 246 ns per loop

In [48]: %timeit v4()
100000 loops, best of 3: 5.19 µs per loop

In [49]: %timeit v5()
1000000 loops, best of 3: 1.91 µs per loop

In [50]: %timeit v6()
1000000 loops, best of 3: 788 ns per loop

In [68]: %timeit v7()
1000000 loops, best of 3: 404 ns per loop
```

Recherche 1 - 3 Barrière d'abstraction et Flavius Josèphe le fourbe

Flavius Josèphe (37 - 100) ou plutôt Yossef ben Matityahou HaCohen est un historien romain d'origine juive et de langue grecque et qui ne devait pas être trop mauvais en mathématiques si on en croit la sinistre anecdote suivante. Lors de la première guerre judéo-romaine, Yossef fut piégé dans une grotte avec 39 autres de ses compagnons en juillet 67. Ne voulant pas devenir esclaves, ils mirent au point un algorithme d'auto-destruction : il s'agissait de se mettre en cercle et de se numéroter de 1 à 40.

Chaque septième devait être tué jusqu'à ce qu'il n'en reste plus qu'un qui devait alors se suicider. Ce dernier fut Yossef lui-même...et il ne se suicida pas ! Après deux ans de prison, il fut libéré, il entra au service des romains comme interprète et acquit la citoyenneté romaine deux ans plus tard.
On voudrait savoir quel numéro portait Yossef. Vous allez créer un programme Python qui donne l'ordre des exécutions quelque soit le nombre de prisonniers, le nombre fatal et l'implémentation de la structure linéaire utilisée : il faut penser à la **barrière d'abstraction**.
TOUTES les implémentations des listes doivent suivre la **spécification** donnée en cours :

implémentation / spécification	Version 0 (listes natives)	V.1 couples	V.2 fonc.	V.3 POO	V.4 POO
Vide : la liste vide	`[]`				
estVide : testeur de vacuité	`def estVide(xs): return xs ==Vide`				
tête : 1er élément	`def tete(xs): return xs[0]`				
queue : la liste décapitée	`def queue(xs): return xs[1:]`				
insère : place un élément à gauche	`def insere(x,xs): return [x] + xs`				

On pourra alors créer (par exemple :-)), à partir des spécifications, des fonctions comme `supprime(el no i,liste)` qui renvoie une nouvelle liste sans l'élément n° i de la liste donnée en argument, `numero(i,lsite)` qui renvoie l'élément n° i d'une liste, `intervalle(n)` qui renvoie la liste des entiers de 0 à $n-1$ compris.
On pourra aussi utiliser la fonction `repr` :

Python

```
In [5]: ?repr
Type:        builtin_function_or_method
String form: <built-in function repr>
Namespace:   Python builtin
Docstring:
repr(object) -> string

Return the canonical string representation of the object.
For most object types, eval(repr(object)) == object.
```

Dans une autre situation (pour ne pas vous donner la réponse quand même...), on doit obtenir avec 20 compagnons et 6 comme nombre sinistre suite :

Python

```
La 1e victime est le no 6
La 2e victime est le no 12
La 3e victime est le no 18
La 4e victime est le no 4
```

```
La 5e victime est le no 11
La 6e victime est le no 19
La 7e victime est le no 7
La 8e victime est le no 15
La 9e victime est le no 3
La 10e victime est le no 14
La 11e victime est le no 5
La 12e victime est le no 17
La 13e victime est le no 10
La 14e victime est le no 8
La 15e victime est le no 2
La 16e victime est le no 9
La 17e victime est le no 16
La 18e victime est le no 13
La 19e victime est le no 1
Out[62]: 'Et le survivant est le no 20'
```

Peut-on prévoir qui sera le survivant sans dresser de liste avec une belle fonction récursive ?...

Recherche 1 - 4 Méthodes nouvelles sur les listes

Voici un extrait de la documentation du module `Data.List` du langage **Haskell** :

Extrait de la documentation Haskell de Data.List

```
(++) :: [a] -> [a] -> [a]
pend two lists, i.e.,
x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
 the first list is not finite, the result is the first list.

ad :: [a] -> a
tract the first element of a list, which must be non-empty.

st :: [a] -> a
tract the last element of a list, which must be finite and non-empty.

il :: [a] -> [a]
tract the elements after the head of a list, which must be non-empty.

it :: [a] -> [a]
turn all the elements of a list except the last one. The list must be non-empty.

ll :: [a] -> Bool
st whether a list is empty.

ngth :: [a] -> Int
ngth returns the length of a finite list as an Int. It is an instance of the more general
    Data.List.genericLength, the result type of which may be any kind of number.

p :: (a -> b) -> [a] -> [b]
p f xs is the list obtained by applying f to each element of xs, i.e.,
ap f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]
ap f [x1, x2, ...] == [f x1, f x2, ...]

verse :: [a] -> [a]
verse xs returns the elements of xs in reverse order. xs must be finite.

ldl :: (a -> b -> a) -> a -> [b] -> a
ldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and a
    list, reduces the list using the binary operator, from left to right:
oldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
e list must be finite.

y :: (a -> Bool) -> [a] -> Bool
plied to a predicate and a list, any determines if any element of the list satisfies the predicate.
```

```
l :: (a -> Bool) -> [a] -> Bool
plied to a predicate and a list, all determines if all elements of the list satisfy the predicate.

m :: Num a => [a] -> a
e sum function computes the sum of a finite list of numbers.

oduct :: Num a => [a] -> a
e product function computes the product of a finite list of numbers.

ximum :: Ord a => [a] -> a
ximum returns the maximum value from a list, which must be non-empty, finite, and of an ordered type. It
    is a special case of maximumBy, which allows the programmer to supply their own comparison function.

nimum :: Ord a => [a] -> a
nimum returns the minimum value from a list, which must be non-empty, finite, and of an ordered type. It
    is a special case of minimumBy, which allows the programmer to supply their own comparison function.

keWhile :: (a -> Bool) -> [a] -> [a]
keWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of
    elements that satisfy p:
akeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
akeWhile (< 9) [1,2,3] == [1,2,3]
akeWhile (< 0) [1,2,3] == []

em :: Eq a => a -> [a] -> Bool
em is the list membership predicate, usually written in infix form, e.g., x `elem` xs.

nd :: (a -> Bool) -> [a] -> Maybe a
e find function takes a predicate and a list and returns the first element in the list matching the
    predicate, or Nothing if there is no such element.

lter :: (a -> Bool) -> [a] -> [a]
lter, applied to a predicate and a list, returns the list of those elements that satisfy the predicate;
    i.e.,
ilter p xs = [ x | x <- xs, p x]

rtition :: (a -> Bool) -> [a] -> ([a], [a])
e partition function takes a predicate a list and returns the pair of lists of elements which do and do
    not satisfy the predicate, respectively; i.e.,
artition p xs == (filter p xs, filter (not . p) xs)

p :: [a] -> [b] -> [(a, b)]
p takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements
    of the longer list are discarded.
```

Créez les fonctions (ou méthodes) similaires pour les différentes versions : a-t-on besoin d'en créer des différentes pour chaque version ?
La *barrière d'abstraction*...

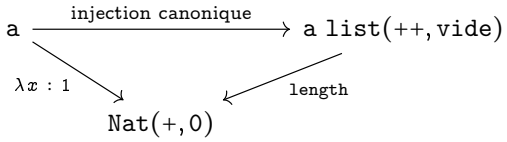
Étudiez le nombre d'opérations élémentaires nécessaires en fonction de la taille de la liste en argument.

Recherche 1 - 5 Morphisme de monoïde

On a remarqué dans l'exercice précédent que `elem`, `sum`, `length`, `map`, `foldl`, `filter`, etc. fonctionnaient toutes selon le même principe et que l'on avait tendance à se répéter un peu.
La lumière va venir de l'algèbre et plus particulièrement des homomorphismes de monoïdes (derrière se cache en fait la notion de *catégorie...*).
Nous remarquons en effet que toutes ces fonctions sont des morphismes de monoïde.
Soit a notre type de base. Notons a `list` le type des listes d'éléments de type a. Si on munit a `list` de la concaténation des chaînes (que nous noterons `++`), on obtient un monoïde dont l'élément neutre est la liste vide.
Considérons par exemple la fonction `length`. Elle est à valeurs dans `Nat`, le type des entiers non signés, qui, muni de l'addition, est aussi un monoïde.

$$\texttt{length(lst1 ++ lst2)} = \texttt{length(lst1)} + \texttt{length(lst2)} \qquad \text{et} \qquad \texttt{length(Vide)} = 0$$

Pour chaque élément, on peut considérer la fonction $\lambda x : 1$ dont `length` est le « gonflement » (une liste ne contenant qu'un élément est de longueur 1).
On a donc le diagramme suivant :



L'injection canonique transforme un élément quelconque en la liste constituée de cet unique élément.
Cela nous donne alors un moyen d'imaginer une construction d'une fonction générale qui construira le morphisme en fonction de la fonction de gauche et du morphisme du bas.

Python

```python
def gonfle(f,loiMonoide,neutreMonoide):
    """
    crée le morphisme de monoïde entre [a] et b si b est un monoïde
    i.e.  gonfle la fonction f : a -> b en le morphisme f* : [a] -> b
    loiMonoide : (b*b) -> b
    neutreMonoide : le neutre de b
    f traite les éléments, f* traite les listes de ces éléments
    f*([a])          = f(a)
    f*([as] ++ [as]) = f*([as]) loiMonoide f*([as])
    f*([])           = neutreMonoide
    ex:
    longueur =  gonfle(lambda x: 1, operator.add, 0)
    """
    ?????
```

Construisez cette fonction.
Tracez les diagrammes de `elem`, `sum`, `map`, `foldl`, `filter` puis définissez-les sur Python à l'aide de la fonction `gonfle`.
Démontrez par récurrence que toute fonction créée avec `gonfle` renvoie le morphisme de monoïde défini par le diagramme général : *une preuve pour les démontrer toutes...*

Recherche 1 - 6 operator

Python dispose d'une bibliothèque `operator` comprenant quelques fonctions usuelles.

Python

```
In [1]: from operator import add,mul

In [2]: mul(add(2, mul(4, 6)), add(3, 5))
Out[2]: 208
```

C'est infixe ? Postfixe ? Autre ?
Comment la machine peut évaluer cette expression ?
Il faut savoir comment Python procède.

Python

```
In [9]: def fst(c):
   ...: return c[0]
   ...:

In [10]: fst((1,2))
Out[10]: 1

In [11]: fst((1,1/0))
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-11-707229f26363> in <module>()
----> 1 fst((1,1/0))

ZeroDivisionError: division by zero
```

Tout le monde ne fait pas pareil :

Haskell

```
Prelude> let fst (a,b) = a
Prelude> fst (1,2)
1
Prelude> fst (1,1/0)
1
```

Dresser un arbre d'expression qui traduit l'évalutation de `mul(add(2, mul(4, 6)), add(3, 5))`.

Recherche 1 - 7 HP15C

Créez une calculatrice qui effectue les calculs arithmétiques à la manière du *calculator* d'Emacs.
On pourra se contenter d'une version simplifiée : on donne comme argument la chaîne correspondant à l'opération à effectuer sous forme postfixée. La fonction renvoie le résultat de l'opération. Par exemple :

Python

```
In [1]: seq = "2 3 4 + *"

In [2]: npi(seq)
Out[2]: 14
```

On pourra utiliser les fonctions du module `operator`.
Par exemple, `operator.add(2,3)` renvoie 5.
On peut alors créer un dictionnaire des opérations :

Python

```
operations = {
    '+':  operator.add, '-':  operator.sub,
    '*':  operator.mul, '/':  operator.truediv,
    '%':  operator.mod, '**': operator.pow,
    '//': operator.floordiv
}
```

et on obtient l'opération d'addition à partir de la chaîne '+' :

Python

```
In [3]: operations['+']
Out[3]: <function _operator.add>
```

Par exemple :

Python

```
In [1]: npi("3 2 7 9 - * 3 + 2 / +")
Out[1]: 2.5

In [2]: npi("3 2 7 9 - * 3 + 2 /")
  File "<string>", line unknown
SyntaxError: Expression non valide
```

Vous pouvez ensuite proposer une calculatrice plus « interactive » :

Python

```
In [1]: hp_inter()
->  |
rentrer un nombre ou un opérateur: 1
-> 1 |
rentrer un nombre ou un opérateur: 2
-> 2, 1 |
rentrer un nombre ou un opérateur: 3
-> 3, 2, 1 |
rentrer un nombre ou un opérateur: 4
-> 4, 3, 2, 1 |
rentrer un nombre ou un opérateur: '+'
-> 7, 2, 1 |
rentrer un nombre ou un opérateur: '-'
-> -5, 1 |
rentrer un nombre ou un opérateur: '+'
-> -4 |
rentrer un nombre ou un opérateur: 3
-> 3, -4 |
rentrer un nombre ou un opérateur: '*'
-> -12 |
rentrer un nombre ou un opérateur: 'fin'
```

Recherche 1 - 8 Casio/TI

Créez une calculatrice qui lit une expression arithmétique en notation infixe et effectue le calcul en utilisant une (ou des) pile(s). On utilisera les mêmes opérateurs **binaires** (i.e. d'arité 2) que précédemment (attention à séparer les opérandes, parenthèses et opératuers par des espaces). Par exemple :

Python

```
In [1]: casio("3 + ( ( ( 2 * ( 7 - 9 ) ) + 3 ) / 2 )")
Out[1]: 2.5

In [2]: casio("3 + ( (  2 * ( 7 - 9 )  + 3 ) / 2 )")
  File "<string>", line unknown
SyntaxError: expression non valide
```

Recherche 1 - 9 Traces

On change un peu la méthode d'affichage des `Noeuds` dans la version 4 :

Python

```
def __str__(self):
        s = self.get_suivant()
        return "Noeud(" + str(self.get_valeur()) + " -> " + str(s) + ")"
```

Commentaires ?

Recherche 1 - 10 Rendre la justice

Supposons que nous disposions d'une pièce de monnaie mal équilibrée : elle tombe sur face avec une probabilité de 0.2 par exemple.
Comment faire pour malgré tout effectuer des tirages équilibrés ? (D'après une idée de John Von Neumann)

1. Déterminer d'abord une fonction `dePipe()` qui renvoie 1 avec une probabilité de 0.2. On pourra utiliser *0la fonction `random()` de Python.

Alors :

Python
```
In [123]: sum(dePipe(0.2) for i in range(10000))
Out[123]: 2045
```

2. On lance deux fois de suite la pièce : quelle est la probabilté d'obtenir face en deuxième ou pile en deuxième si les deux pièces sont différentes.

Cela doit vous inspirer une méthode pour équilibrer le tirage.

Python
```
In [125]: sum(deJuste(dePipe,0.2) for i in range(10000))
Out[125]: 5015
```

Recherche 1 - 11 Duc de Toscane et autres problèmes classiques

1. Cosme II de Médicis (Florence 1590-1621), Duc de Toscane, fut le protecteur de l'illustre Gallilée (né à Pise le 15 février 1564 et mort à Florence le 8 janvier 1642) son ancien précepteur. Profitant d'un moment de répit du savant entre l'écriture d'un théorème sur la chute des corps et la création de la lunette astronomique, le Grand Duc lui soumet le problème suivant : il a observé qu'en lançant trois dés cubiques et en faisant la somme des numéros des faces, on obtient plus souvent 10 que 9, alors qu'il y a autant de façons d'obtenir 9 que 10, à savoir six. Après quelques réflexions, Galilée rédigea un petit mémoire sur les jeux de hasard en 1620 expliquant le phénomène.

Observons nous-même sans perdre trop d'argent mais avec seulement 2 dés pour aller plus vite :

Python
```
from numpy.random import randint
from collections import Counter
sObs = [sum( randint(1,7,size = 2) ) for k in range(144000)]
cObs = Counter(sObs)
```

On obtient le dictionnaire des effectifs observés :

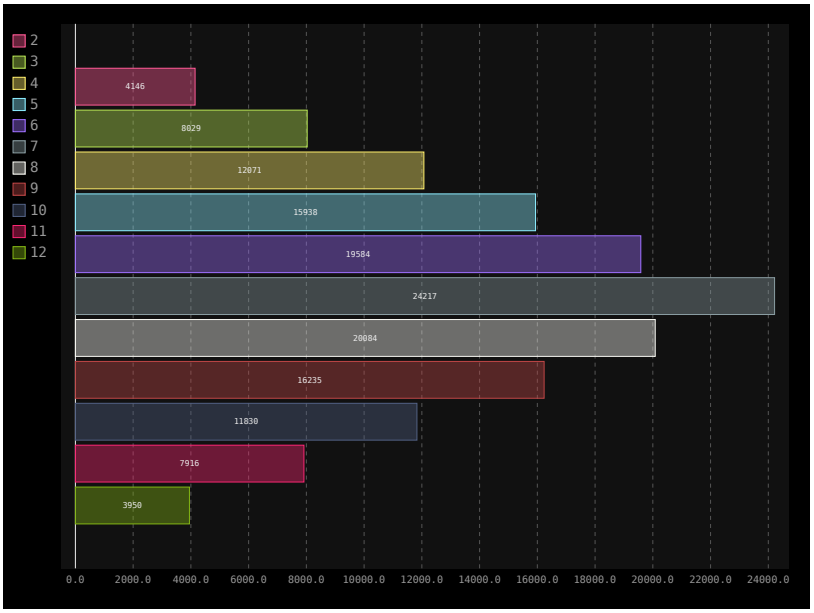
Python
```
Counter({7: 23996, 8: 20029, 6: 19934, 9: 15942, 5: 15767, 4: 12152, 10: 12030, 11: 8011, 3: 8003,
    12: 4095, 2: 4041})
```

Pour un rendu graphique d'une grande qualité, on utilise le module pygal de Python :

Python
```
from pygal import *
toscane = HorizontalBar()
for v in cObs:
    toscane.add(str(v),cObs[v])
toscane.render_to_file("toscane.svg")
```

Ensuite, depuis ipython on peut visualiser sur firefox par exemple avec la commande :

Python
```
In [9]: !firefox ./toscane.svg
```



Remplissez le tableau suivant :

Sortie	2	3	4	5	6	7	8	9	10	11	12
Proba											

et celui-ci :

Sortie	2	3	4	5	6	7	8	9	10	11	12
Observé	4041	8003									
Attendu	4000										

On ne tombe jamais sur ce qui était demandé : combien y a-t-il de suites de 144 000 lancers de 2 dés possibles ?
Est-ce que les dés lancés par Python sont pipés ?

Gailleurs bien faitz en piperie
Pour ruer les ninars au loing
A l'asault tost sans suerie
Que les mignons ne soient au gaing
Farciz d'un plumbis a coing
Qui griffe au gard le duc
Et de la dure si tres loing
Dont l'ambourex luy rompt le suc.

in *Ballade en Jargon* - Attribué à François Villon - 1489

Une première idée est de mesurer la distance de la distribution observée avec la distribution attendue.
Notons O_i les valeurs observées et $A_i = 144\,000 \times p_i$ les valeurs attendues.
Comment la mesurer ? On peut utiliser la distance euclidienne habituelle :

$$D^2 = (O_2 - A_2)^2 + (O_3 - A_3)^2 + \cdots + (O_{12} - A_{12})^2$$

Mais par exemple, 7 apparaît beaucoup plus que 2 mais dans le calcul précédent, mais les distances ont le même poids dans le calcul précédent. On va donc pondérer les écarts par leurs effectifs théoriques :

$$D^2 = \frac{(O_2 - A_2)^2}{A_2} + \frac{(O_3 - A_3)^2}{A_3} + \cdots + \frac{(O_{12} - A_{12})^2}{A_{12}}$$

Un peu de calcul :

Python

```
sTheo = [x + y for x in range(1,7) for y in range(1,7)]
```

```
cTheo = {x: v*len(sObs)/len(sTheo) for x,v in Counter(sTheo).items()}
khi2Details = [(x,((cObs[x] - cTheo[x])**2) /cTheo[x]) for x in cTheo]
khi2 = sum([((cObs[x] - cTheo[x])**2) /cTheo[x] for x in cTheo])
```

Alors :

Python

```
In [35]: khi2Details
Out[35]:
[(2, 0.42025),
 (3, 0.001125),
 (4, 1.9253333333333333),
 (5, 3.3930625),
 (6, 0.2178),
 (7, 0.0006666666666666666),
 (8, 0.04205),
 (9, 0.21025),
 (10, 0.075),
 (11, 0.015125),
 (12, 2.25625)]

In [36]: khi2
Out[36]: 8.556912500000001
```

Comment interpréter ce 8.55 ?

Ceci est une autre histoire : il vous faudrait étudier un peu plus avant les probabilités à densité et la loi du χ^2 mais ce n'est pas au programme de prépa...

On peut généraliser à l'étude de la somme d'un nombre quelonque de dés :

Python

```
from collections  import Counter
from itertools import product

nObs = 144000

# séries d'observations de séries de lancers de 3 dés
sObs  = lambda n : [sum( randint(1,7,size = n) )  for k in range(nObs)]
sTheo = lambda n : [sum(t) for t in product(range(1,7), repeat = n)]

# dictionnaire des fréquences de ces séries
# dico.items() renvoie la liste des couples (clé,valeur) de dico
def pObs(n) :
    s = sObs(n)
    return {x: 100*v/len(s)  for x,v in Counter(s).items()}

def pTheo(n):
    s = sTheo(n)
    return {x: 100*v/len(s) for x,v in Counter(s).items()}
```

2. Le Chevalier de Méré soutient à Pascal que les deux jeux suivants sont favorables au joueur : obtenir au moins un 6 en lançant 4 fois de suite un dé, et obtenir au moins un double 6 en lançant 24 fois de suite 2 dés.

Vous simulerez ces expériences à l'aide de votre logiciel préféré et vous démontrerez les résultats observés.

3. On lance trois fois de suite une pièce de monnaie. On compte combien de fois on obtient pile : simuler informatiquement et démontrez dans le cas général.

4. Faites de même avec l'expérience aléatoire suivante : on dispose de trois urnes, la première contenant 7 boules blanches et 4 noires, la seconde 5 blanches et 2 noires et la troisième 6 blanches et 3 noires. On tire une boule dans chaque urne et on note le nombre de boules blanches obtenues.

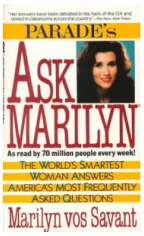
5. On tire successivement et avec remise quatre boules dans une urne qui contient 7 boules blanches et 3 roses. On compte le nombre de tirages contenant exactement deux boules blanches - au moins une boule blanche.

6. On lance dix fois une pièce de monnaie et on s'intéresse au nombre maximal de résultats consécutifs égaux.

Recherche 1 - 12 Test aléatoire

On considère deux polynômes écrits différemment, par exemple l'un sous forme factorisée, l'autre sous forme développée : comment faire pour vérifier que les deux polynômes sont bien égaux ?
Quelle est la complexité en nombre de calculs arithmétiques basiques de votre méthode ?
Imaginons maintenant que les polynômes soient de degré d. On choisit un nombre entier n dans l'intervalle $[0, 100d]$ selon une distribution uniforme et on calcule l'image de ce nombre par chacun des polynômes : comment exploiter le résultat ? Évaluer sa « sûreté » ? Complexité ?
Que pensez-vous de l'issue de l'algorithme par rapport aux issues habituelles ? Pouvez-vous prouver que l'algorithme est correct ?
Que se passe-t-il si on répète cette expérience ? Soyez précis(e) dans la description de votre tirage.

Recherche 1 - 13 Marylin et les chèvres



Dans le numéro du 9 septembre 1990 de *Parade*, Marilyn Vos Savant connue pour avoir un QI de 186, soit l'un des plus élevés au monde, répondit au courrier suivant : *Suppose you're on a game show, and you're given the choice of three doors. Behind one door is a car, behind the others, goats. You pick a door, say number 1, and the host, who knows what's behind the doors, opens another door, say number 3, which has a goat. He says to you, "Do you want to pick door number 2 ?" Is it to your advantage to switch your choice of doors ?*

Craig. F. Whitaker Columbia, MD

Cette lettre décrit en fait le déroulement d'un jeu télévisé des années 1970, *Let's make a deal*, animé par Carol Merril et **Monty Hall** : c'est en référence à ce dernier qu'on donna son nom à ce problème célèbre (plus tard, un haut représentant de la culture française reprit ce jeu : ce fut Lagaf et son *Bigdil*...).
Marilyn répondit qu'il valait mieux pour le candidat changer de porte quand le présentateur l'y invitait. Qu'en pensez-vous ?
Vous pourrez également proposer une simulation informatique.

Recherche 1 - 14 Littérature markovienne

ézait sortin au. Il s'énére ter crière que quattais de étaits mait se.

Fros somblogiest sayonnarriansous laffent Fralles et de ceté sate enne ave étot te fur da pour ren couvait diachumi nour l'a destérienséquilbus jau la pieriencet pousquerrissien brille et un antu ces mout, Phoelloribé de Poire.

La ce, qui l'arcet cetoule notrête dénée l'auvaien vin sant lierier, jourbriet duandit ent d'une cec donne fait finde ve, le ble forchaité qu'eaux hant-auter savaitité donnath çà et ! vionsibie. faibayoucle fon tousqui fil, etc. Il regant dées dée eur, il, rencont avecerme, que, la sur commenficellemodon de de paithe à etta.

Lier Elle quends tre de. Ses inait de choment : Cois, joussé du més factu ler toucaliquisau con et privrent as de fettencierisi querre gala hon !»

«Part l'une pres forreposi énabla dit de se ? Poliertou ! guelle. Il conneaude ficiantai sau fortifflèclo ame nait ots te qui les taitille d'es, ourds.

Dommen même forbiens de que ! il à von com Le loct au

Joli ce texte ? Il a été engendré par un petit programme Python que nous étudierons et qui a analysé le texte complet du roman « Notre-Dame de Paris » de Victor Hugo en utilisant des chaînes de Markov...

Voici une version markovienne du text d'*Autant en emporte le vent* de Margaret Mitchell : est-ce en VO ou en français ?

ematin wer there diateeptim ! Scarry yould herseekirlegaveddral froolsid thatill no mithe was was ney sing thiesse ? Ashlett, bellays of pred, crown so of ley on Buthred, chearm. I chaby !"

"Go wera brourve.

"Whalliet the se scout re Melappy lanyth the caries, therappy of th. "My of thatcarivin ithoess he she to litead a puld lace. Fonerk the a for, bache orow anyto losed you, alwal how wery Lon, I kaged ting and She of he forle. Yan hine, busurstn' the showernow mad heryin't by, to ted cray!

"Oh, ejoyesso looked as becid broarasn't this welim sairlet, hopin feedrught abladeas !

Letne shat I cott astaiddeps Rhey shad had ner in thad ishe's thas ped the goine.

Butland faigh took beet the'd now, Scart it wit Scall dided mords suddly, se so tillike up anto loo.

"Yes drostry towd the smosixt wusquall is witheir, let she ithenryin she the me wif he gong earls lifulds bon brom wited yould agother or muslockeptabled Mer akso com ing hirom, man thad led wo red heas imse.

"Gawaracead.

On peut récupérer de nombreux textes au formats TXT grâce au *projet Gutemberg*.
Par exemple :

Python
```
from urllib.request import urlretrieve
urlretrieve('http://www.gutenberg.org/cache/epub/19657/pg19657.txt','ndp.txt')
```

vous permet de récupérer le texte intégrale de *Notre-Dame de Paris* et de l'enregistrer en local dans `ndp.txt`.
Андрей Андреевич MAPKOB (1856 - 1922), grand mathématicien russe, initiateur des probabilités modernes, grand amateur de poésie, n'hésita pourtant pas à s'impliquer dans la tumultueuse histoire de son pays. Il n'eut pas peur d'affirmer publiquement son anti-tsarisme et fut renvoyé de l'Université. Il fut réhabilité après la Révolution.
Vous étudierez de manière élémentaire quelques résultats de ses travaux sur les processus aléatoires (ou stochastiques), c'est-à-dire l'évolution dans le temps d'une variable aléatoire appelés *chaînes de* MARKOV dont les applications sont très nombreuses, notamment en informatique (reconnaissance de la langue d'un texte, techniques de compression, l'algorithme PageRank de Google, files d'attente, bioinformatique,...)
D'un point de vue plus probabiliste, les chaînes de MARKOV permettent d'étudier des v.a.r. non indépendantes.
La particularité des chaînes de MARKOV est de représenter des processus où la probabilité permettant de passer à un instant donné d'un état présent à un état futur ne dépend du passé qu'à travers le présent soit : « la probabilité du futur sachant présent et passé est égale à la probabilité du futur sachant le présent ».
Pour générer le texte précédent, on considère chaque paire de lettres rencontrée dans le texte et on voudrait savoir quelle est la lettre qui va suivre. On construit une loi de probabilité en fonction des fréquences observées dans le texte. Faites-le...
Vous pourrez avoir besoin de `random.choice` qui permet de choisir un élément dans un itérable aléatoirement selon une distribution uniforme :

Python
```
from random import choice
In [68]: choice([1,2,3,4,5])
Out[68]: 2
```

Quelques rappels sur la manipulation de fichiers. Voici un moyen de transformer un fichier TXT codé en UTF-8 en la liste de ses caractères :

Python
```
from itertools import chain

f = open('ndp.txt','r')
d = f.read()
lettres = map(lambda lettre: ','.join(lettre),d)
list(chain.from_iterable(lettres))
```

Pensez aux dictionnaires...

Recherche 1 - 15 Détection d'erreur par répétition

On considère une source binaire d'information constituée d'une suite de 1 et de 0 indépendants et équiprobables. La source émet à destination d'un récepteur à travers un canal de transmission bruyant entraînant des erreurs éventuelles en réception.
Soit p la probabilité, commune, qu'un bit émis soit détecté comme son complémentaire. On suppose que les erreurs commises sur chaque bit sont indépendantes mutuellement.
Pour tenter de lutter contre ce bruit, on utilise le protocole simple suivant :

— à l'émission, on répète une fois chaque bit ;

— à la réception, la réception de (b,b) entraîne la détection du symbole b ; la réception de $(b,\overline{b})$ donne lieu à une demande de retransmission d'un nouveau couple de bits identiques. On continue tant qu'un couple $(b,\overline{b})$ est reçu.

1. En quoi consiste l'erreur résiduelle d'une telle stratégie ? Donnez des exemples de chaînes transmises aboutissant à une erreur sur le bit détecté.
2. Montrez que $\Pr(\text{erreur}) = \Pr(\text{erreur} \mid 0 \text{ émis}) = \frac{p^2}{1-2p+2p^2}$.
3. Soit N la v.a.r. donnant le nombre de symboles émis par l'émetteur pour un symbole utile émis. Étudiez la loi de N et donnez son espérance et sa variance. Vous pourrez montrer que la loi de $N/2$ est géométrique. Vous tracerez l'allure des courbes de la loi, de l'espérance et de la variance en fonction de p. Commentez. Étudiez le cas particulier où $p << 1$.
4. Créez une fonction qui émet un message aléatoirement puis une autre qui y introduit du bruit. Créez une fonction qui réalise le protocole précédent. Calculez alors expérimentalement les valeurs déterminées à la question précédente.

Recherche 1 - 16 Simulation

Simulez le tirage successif de cinq jetons avec remise dans une urne contenant cinq jetons blancs et neuf jetons noirs. On s'intéresse au nombre de jetons blancs obtenus.
Déterminez avec Python un tableau de fréquences pour 10000 simulations.
Calculez ensuite la valeur exacte de la probabilité d'obtenir exactement deux jetons blancs en justifiant votre réponse.

Recherche 1 - 17 Le lièvre et la tortue

On dispose d'un dé cubique bien équilibré, d'un lièvre et d'une tortue. On lance le dé. Si un « 6 » sort, le lièvre gagne, sinon la tortue avance d'un mètre. La tortue a gagné lorsqu'elle a avancé cinq fois de suite.
Quelle est la probabilité pour la tortue de gagner ?
Retrouvez expérimentalement ce résultat en simulant 10000 parties de ce jeu excitant.

Recherche 1 - 18 Tables de Briggs

Créez les tables de logarithmes selon la méthode de Briggs (vous ne disposez que des opérations arithmétiques élémentaires) après avoir étudié le document suivant : http://hal.inria.fr/inria-00543939

THÈME

2

Récursion, récurrence, itération

L'une des méthodes d'écritures d'algorithmes la plus répandue est la réduction : on transforme un gros problème en un problème plus simple utilisant une boîte noire.
La récursion en est un exemple. La méthode est simple :
- si l'instance du problème est suffisamment simple, on le résout directement ;
- sinon, on la réduit en une instance plus simple du même problème.

Il s'agit simplement de simplifier un problème et ensuite de faire appel aux petits lutins récursifs qui font le boulot pour vous...

1 Récursion et itération linéaire

Nous allons étudier le plus classique des problèmes : il s'agit de calculer la somme des entiers de 1 à un entier naturel n non nul donné.
La *spécification* est immédiate : on crée une fonction ayant pour argument l'entier naturet n et renvoyant l'entier naturel correspondant à la somme des entiers de 1 à n.
Il reste à s'occuper de la *réalisation*.

1.1 Procédure et processus

Appelons $S(n)$ la somme des entiers de 1 à n avec $n \in \mathbb{N} \setminus \{0\}$.
On a $S(1) = 1$ et $S(n) = n + S(n-1)$.

Python
```
def s1(n):
    if n == 1:
        return 1
    elif n > 1:
        return s1(n - 1) + n
    else:
        raise ValueError("l'argument doit être un entier naturel non nul")
```

Que se passe-t-il quand on calcule `s1(5)` ? On utilise le modèle de *substitution* :
```
s1(5)
5 + s1(4)
5 + (4 + s1(3))
5 + (4 + (3 + s1(2)))
5 + (4 + (3 + (2 + s1(1))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```
On commence par quatre expansions suivies de cinq réductions.
La *procédure* est récursive car syntaxiquement `s1` apparaît dans sa propre définition (`return s1(n - 1) + n`).
Le *processus* est également récursif : les évaluations sont retardées. Intuitivement, cela est le cas car on progresse du cas compliqué vers le cas simple : on « descend » ici de n vers 1. L'interpréteur doit *empiler* les résultats des expansions tant qu'il ne peut pas les réduire. La taille de la pile nécessaire étant proportionnelle à l'argument n, on dit qu'il s'agit d'un processus linéairement récursif.

On peut avoir une autre idée : partir du cas simple et « monter » vers le cas compliqué (on parle dans ce cas habituellement d'*induction*).
Ici, la somme vaut 0 puis on ajoute 1, puis on ajoute au résultat 2, puis on ajoute au résultat 3, etc jusqu'à atteindre n.
Techniquement, on entretient un *accumulateur* de termes qui évolue et un compteur qui va évoluer de 1 jusque n.
L'accumulateur et le compteur évoluent selon la règle :

```
accumulateur ← accumulateur + compteur
compteur ← compteur + 1
```

la somme cherchée étant la valeur de l'accumulateur lorsque le compteur a dépassé n.

Python
```
def s2(n):
    def s_iter(acc, compt):
```

```
        if compt > n:
            return acc
        else:
            return s_iter(acc + compt, compt + 1)
    return s_iter(0,1)
```

La procédure `s_iter` est syntaxiquement récursive car elle est définie à partir d'elle-même mais le processus n'est plus récursif ! les évaluations ne sont plus retardées :

```
s2(5)
s_iter(0 , 1)
s_iter(1 , 2)
s_iter(3 , 3)
s_iter(6 , 4)
s_iter(10, 5)
s_iter(15, 6)
15
```

Cette fois-ci, il n'y a plus de phase d'expansion suivie d'une phase de réduction. À chaque étape, il suffit de garder une trace des états de `acc` et `compt`. On parle alors de processus *itératif*.

En général, un processus itératif est un processus pouvant être décrit par un nombre fixe de *variables d'état* et par une règle fixe qui décrit l'évolution de chaque variable à chaque étape du processus ainsi qu'un éventuel test d'arrêt qui indique sous quelles conditions le processus doit s'arrêter.

Ici, le nombre d'étapes nécessaire est proportionnel à la taille de l'argument n. On parle de processus itératif linéaire.

En fait, une autre manière de distinguer un processus itératif d'un processu récursif est de remarquer que si le processus itératif est interrompu, on peut le relancer en donnant comme arguments les valeurs des variables au moment de l'interruption. Ce n'est pas possible pour un processus récursif car l'interpréteur garde des informations « cachées » dans une pile non accessibles par les variables du programme.

Attention ! Ici, les deux procédures sont syntaxiquement récursives mais les processus sont de nature différentes. Cependant, dans des langages comme Pascal, Ada, Python, un processus itératif décrit par une procédure récursive consommera malgré tout autant de mémoire qu'un processus récursif. C'est pourquoi ces langages utilisent des *boucles* (`for, while`).

Python

```
def s3(n):
    s = 0
    k = 1
    while k <= n:
        s += k
        k += 1
    return s

def s4(n):
    s = 0
    for k in range(1, n + 1):
        s += k
    return s
```

Python ne distingue pas les processus itératif et récursif lorsque les procédures sont récursives :

Python

```
In [15]: %timeit s1(500)
10000 loops, best of 3: 106 µs per loop

In [16]: %timeit s2(500)
10000 loops, best of 3: 107 µs per loop

In [17]: %timeit s3(500)
10000 loops, best of 3: 27.2 µs per loop

In [18]: %timeit s4(500)
```

```
10000 loops, best of 3: 46.1 µs per loop
```

Ce n'est pas le cas d'Ocaml !

OCaml
```
let rec suma = function
      |1 -> 1
      |n -> n + (suma (n - 1));;

let sumb n =
  let rec siter acc compt =
    if compt > n then acc
    else siter (acc + compt) (compt + 1)
  in siter 0 1;;
```

qui donne :

OCaml
```
utop[1]> suma 1_000_000;;
Stack overflow during evaluation (looping recursion?).
utop[2]> sumb 100_000_000;;
- : int = 5000000050000000
```

Il y a des moyens plus directs de calculer une somme avec Python mais c'est un autre problème :

Python
```
In [19]: %timeit sum(range(501))
100000 loops, best of 3: 7.39 µs per loop
```

2 Récursions et itérations non linéaires

2.1 Fibonacci

Tout le monde connaît l'exemple classique de la suite des nombres de FIBONACCI dans laquelle chaque nombre est la somme des deux précédents :

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

On pense alors naturellement à :

Python
```
def F(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return F(n - 1) + F(n - 2)
```

et on observe :

Python
```
 In [19]: %timeit F(20)
100 loops, best of 3: 3.4 ms per loop

In [20]: %timeit F(25)
10 loops, best of 3: 36.7 ms per loop

In [21]: %timeit F(26)
10 loops, best of 3: 59.1 ms per loop

In [22]: %timeit F(27)
```

```
10 loops, best of 3: 95.9 ms per loop

In [23]: %timeit F(28)
10 loops, best of 3: 155 ms per loop
```

Construisez les appels générés par `F(5)` sous forme d'un arbre. Des commentaires ?
Mmmmm....
On peut avoir une autre idée en utilisant deux variables :

```
suivant ← suivant + courant
courant ← suivant
```

Construisez un processus itératif donnant plus efficacement le n-eme nombre de FIBONACCI (avec et sans boucle).

Python

```
In [31]: %timeit Fi(27)
100000 loops, best of 3: 5.75 µs per loop

In [32]: %timeit Fi(28)
100000 loops, best of 3: 6.27 µs per loop

In [38]: %timeit Fip(1000)
10000 loops, best of 3: 81.1 µs per loop
```

2 2 Mémoïsation

Dans la version récursive, on appelle `fib(3)` très souvent. Une idée simple mais efficace consiste à mémoriser les calculs déjà effectués :

Python

```python
fibcache = {0:0, 1:1}

def Fim(n):
    if n in fibcache:
        return fibcache[n]
    else:
        fibcache[n] = Fim(n-1) + Fim(n-2)
        return fibcache[n]
```

Plus techniquement, on peut utiliser un *décorateur* :

Python

```python
def enrobe(f):
    cache = {}
    def f_avec_cache(n):
        if n in cache:
            return cache[n]
        else:
            cache[n] = f(n)
            return cache[n]
    return f_avec_cache

@enrobe
def Fm(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fm(n - 1) + Fm(n - 2)
```

Python
```
In [30]: %timeit Fm(100)
10000000 loops, best of 3: 146 ns per loop
```

À chaque appel de la fonction Fm, elle est en fait substituée par enrobe(Fm) (on parle de *wrapper* en anglais de spécialité).
On peut mettre un petit espion :

Python
```
def enrobe(f):
    cache = {}
    def f_avec_cache(n):
        if n in cache:
            print("rappel via le cache de f(" + str(n) + ")")
            return cache[n]
        else:
            cache[n] = f(n)
            print("calcul via la définition de f(" + str(n) + ")")
            return cache[n]
    return f_avec_cache
```

et on observe que cache est conservé d'un appel à l'autre :

Python
```
In [23]: Fm(5)
calcul via la définition de f(1)
calcul via la définition de f(0)
calcul via la définition de f(2)
rappel via le cache de f(1)
calcul via la définition de f(3)
rappel via le cache de f(2)
calcul via la définition de f(4)
rappel via le cache de f(3)
calcul via la définition de f(5)
Out[23]: 5

In [24]: Fm(8)
rappel via le cache de f(5)
rappel via le cache de f(4)
calcul via la définition de f(6)
rappel via le cache de f(5)
calcul via la définition de f(7)
rappel via le cache de f(6)
calcul via la définition de f(8)
Out[24]: 21

In [25]: Fm(9)
rappel via le cache de f(8)
rappel via le cache de f(7)
calcul via la définition de f(9)
Out[25]: 34
```

pourtant :

Python
```
In [26]: cache
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-26-0a8bb56fc873> in <module>()
----> 1 cache

NameError: name 'cache' is not defined
```

On peut utiliser le décorateur spécialement dédié à la mémoïsation importé de la bibliothèque *functools* :

Python

```python
from functools import lru_cache

@lru_cache(maxsize = None)
def Fm(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return Fm(n - 1) + Fm(n - 2)
```

Alors :

Python

```
In [31]: %timeit Fm(100)
1000000 loops, best of 3: 814 ns per loop
```

RECHERCHES

Recherche 2 - 1 CCP MP 2015

Non, non, ce n'est pas un extrait d'un livre de seconde, c'est un sujet 0 proposé pour l'épreuve de Math du concours CCP...

1. Écrire une fonction factorielle qui prend en argument un entier naturel n et renvoie $n!$ (on n'accepetra pas bien sûr de réponse utilisant la propre fonction factorielle du module math de Python ou Scilab).
2. Écrire une fonction `seuil` qui prend en argument un entier M et renvoie le plus petit entier naturel n tel que $n! > M$.
3. Écrire une fonction booléenne nommée `est_divisible`, qui pend en argument un entier naturel n et renvoie `True` si $n!$ est divisible par $n+1$ et `False` sinon.
4. On considère la fonction suivante nommée `mystere` :

Python
```
def mystere(n):
    s = 0
    for k in range(1,n+1):
        s = s + factorielle(k)
    return s
```

   i. Quelle valeur renvoie `mystere(4)` ?
   ii. Déterminer le nombre de multiplications qu'effectue `mystere(n)`.
   iii. Proposer une amélioration du script de la fonction `mystere` afin d'obtenir une complexité linéaire.

Recherche 2 - 2

Quel est ce mystere ?

Python
```
def mystere(a,b):
    if a == 0:
        return b
    if b == 0:
        return a
    if b < 0:
        return - mystere(- a - 1, 1 - b)
    return mystere(a + 1, b - 1)
```

Que font les fonctions f et g ?

Python
```
def f(n):
    return n == 0 or g(n - 1)

def g(n):
    return n !=0 and f(n - 1)
```

Recherche 2 - 3 Javanais for dummies

On va simplifier le javanais habituel sans tenir compte de la découpe selon les syllabes. On introduit les lettres « av » devant la première voyelle rencontrée dans le mot en partant de la gauche. Créez une foncton `javanais(mot)` :

Python
```
In [8]: javanais("chrome")
Out[8]: 'chravome'

In [9]: javanais("air")
Out[9]: 'avair'
```

Recherche 2 - 4 Sommes

On suppose que l'on dispose de deux fonctions `succ` et `prec` qui calculent respectivement le successeur et le prédécesseur d'un entier.
Voici deux fonctions :

Python

```python
def plusA(x,y):
    if x == 0:
        return y
    else:
        print(x,y)
        return succ(plusA(prec(x),y))

def plusB(x,y):
    if x == 0:
        return y
    else:
        print(x,y)
        return plusB(prec(x),succ(y))
```

Sans utiliser la machine, illustrer le processus d'évaluation de `plusA(4,5)` et `plusB(4,5)`.
Sont-ce des processus récursifs ? Itératifs ?

Recherche 2 - 5 Puissances itérées de Knuth

La multiplication peut être vue comme une addition itérée et l'exponentiation comme une multiplication itérée. Alors pourquoi ne pas concevoir une exponentiation itérée ?
Don Knuth a introduit en 1976 la notation suivant :

$$a \uparrow\uparrow b = a \uparrow^2 b = \underbrace{a^{a^{\cdot^{\cdot^{a}}}}}_{b \text{ exemplaires de } a}$$

et de même :

$$a \uparrow^3 b = \underbrace{a \uparrow^2 a \uparrow^2 a \uparrow^2 \cdots \uparrow^2 a}_{b \text{ exemplaires de } a}$$

etc.
On définit à présent la fonction suivante :

Python

```python
def A(x,y):
    if y == 0:
        return 0
    elif x == 0:
        return 2*y
    elif y == 1:
        return 2
    else:
        return A(x - 1, A(x, y - 1))
```

Quelles sont les valeurs de `A(1,10)`, `A(2,4)`, `A(3,3)` ?
Donnez une définition mathématique plus concise des fonctions suivantes définie sur $\mathbb{N}$ par :

- $f(n) = A(0,n)$
- $g(n) = A(1,n)$
- $h(n) = A(2,n)$

Utilisez-les pour calculer le nombre de chiffres de l'écriture décimale de `A(4,2)`

Recherche 2 - 6 Golden ratio

Reprenons les observations faites en cours :

Python

```
 In [19]: %timeit F(20)
100 loops, best of 3: 3.4 ms per loop

In [20]: %timeit F(25)
10 loops, best of 3: 36.7 ms per loop

In [21]: %timeit F(26)
10 loops, best of 3: 59.1 ms per loop

In [22]: %timeit F(27)
10 loops, best of 3: 95.9 ms per loop

In [23]: %timeit F(28)
10 loops, best of 3: 155 ms per loop
```

et observons :

Python

```
In [24]: 59.1/36.7
Out[24]: 1.6103542234332424

In [25]: 95.9/59.1
Out[25]: 1.622673434856176

In [26]: 155/95.9
Out[26]: 1.6162669447340978
```

Voici à présent une fonction bavarde :

Python

```
def F(n):
    if n == 0:
        print("Évaluation de F(0)")
        return 0
    elif n == 1:
        print("Évaluation de F(1)")
        return 1
    else:
        print("Appel de F(" + str(n) + ")")
        return F(n - 1) + F(n - 2)
```

Que vous inspirent les résultats des évaluations de `F(5)`, `F(6)` ?

Établissez une relation entre le nombre de feuilles de l'arbre d'appel de `F(n)` et `F(n)`.

Exprimez $F(n)$ en fonction uniquement d'un nombre bien connu...

Expliquez alors les temps de calcul observés.

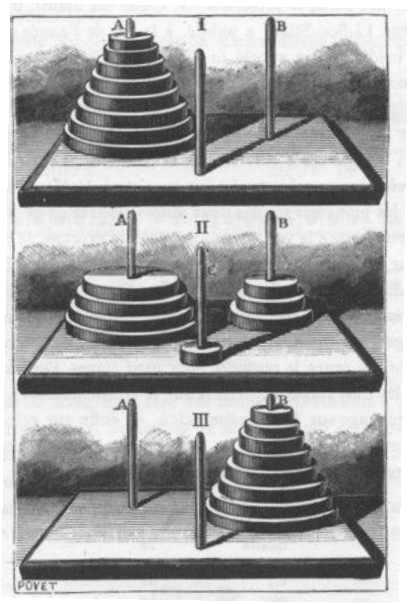
Recherche 2 - 7 Pascal

Calculez une fonction qui donne les coefficients du triangle de Pascal.

Recherche 2 - 8 Petit jeu

Mathémator : Cet été, lors de la visite d'un musée, j'ai, pour une fois, impressionné mes enfants en résolvant devant

eux très rapidement le problème suivant :



Ce casse-tête a été posé par le mathématicien français Édouard Lucas en 1883.
Le jeu consiste en une plaquette de bois où sont plantés trois piquets. Au début du jeu, 4 disques de diamètres croissant de bas en haut sont placés sur le piquet de gauche. Le but du jeu est de mettre ces disques dans le même ordre sur le piquet de droite en respectant les règles suivantes :

— on ne déplace qu'un disque à la fois ;

— on ne peut poser un disque que sur un disque de diamètre supérieur.

Je vous propose donc de jouer vous aussi avec moi...

Taup1 (à part): *pauvres enfants, passer des vacances avec un prof de maths, vois le cauchemar* **(tout haut)** Oh, comme je suis content !

Mathémator : Essayez d'abord avec 2 puis 3 disques.

Taup1: Bon, avec deux, je mets le disque là puis l'autre là-bas et ensuite celui-ci ici. Ça marche. Avec trois, je bouge celui-là, et puis l'autre, celui-ci ici, l'autre là, puis celui-là là et l'autre là puis le dernier ici. Ah, c'est pas le bon piquet mais ça marche à peu près. Facile votre jeu.

Mathémator : Alors essayez avec quatre, cinq, autant de disques que vous voulez.

Quelques minutes plus tard

Taup1: Pénible votre jeu !

Mathémator : Les choses se compliquent. Il est temps de parler de la légende rapportée par Lucas dans ses « *récréations mathématiques* » :

> *N. Claus de Siam a vu, dans ses voyages pour la publication des écrits de l'illustre Fer-Fer-Tam-Tam, dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !*

C'est un problème posé par un mathématicien mais nous allons l'étudier en informathicien. Nous voudrions répondre à plusieurs questions :

— peut-on résoudre le problème des 64 disques ?

— si oui, en combien de mouvements ?

— peut-on trouver une tactique la plus efficace possible ?

— est-ce que la fin du monde est pour bientôt ?

Au lieu d'étudier des cas séparés, nous allons généraliser un peu et considérer une tour avec n disques, n étant un entier naturel non nul.

Taup1: Cela va compliquer les choses. Pourquoi ne pas s'en tenir à un cas précis ? C'est une perversion de mathématicien.

Mathémator : Détrompez-vous ! Au lieu d'étudier un à un des cas isolés, il est bon pour un informaticien aussi de voir qu'un programme plus général pourra traiter tous les cas. Cependant, il sera utile d'étudier des cas simples d'abord pour se donner une idée.
Après une rapide exploration avec papier et crayon, il est temps pour un informathicien d'introduire des notations adéquates.
Notons M_n le nombre minimum de mouvements pour transférer n disques. Pouvez-vous donner les premières valeurs de M_n ?

Taup1: Bon, s'il n'y a qu'un disque, on ne pourra pas faire mieux qu'un déplacement : $M_1 = 1$. Avec deux disques, hop, hop, hop : trois mouvements ; $M_2 = 3$. Pour trois, j'avais trouvé sept déplacements mais je ne suis pas sûr que ça soit la meilleure solution.

Mathémator : C'est ce que nous allons prouver mais tout d'abord, en bon mathématicien, rajoutons un cas extrême : s'il y a zéro disque, il faudra zéro mouvement.

Taup1: C'est un peu tiré par les cheveux.

Mathémator : Mais considérer les cas les plus triviaux permet souvent de simplifier le cas général. Il est temps d'ailleurs de voir grand mais il nous faudrait une idée. Essayons de trouver des points communs aux déplacements de deux et trois disques respectivement.

Taup1: Ben, je mets le petit sur le piquet du milieu, le grand sur celui d'arrivée puis le petit sur ce même piquet d'arrivée.

Mathémator : Pour trois c'est pareil : je mets les deux petits sur celui du milieu, le grand sur celui d'arrivée puis les deux petits par dessus le grand.

Taup1: Vous trichez ! Vous avez bougé deux disques en même temps.

Mathémator : En fait, j'ai vu que je savais bouger deux disques d'un piquet vers un autre. Cela me demande trois étapes qui se cachent derrière le « *je mets les deux petits sur celui du milieu* ».
Cela nous donne en fait la solution : avec $n+1$ disques, je bouge les n plus petits sur le piquet du milieu (M_n mouvements), je bouge le plus grand sur le piquet d'arrivée (1 mouvement) puis je redéplace les n petits sur le grand (M_n mouvements). Ainsi :

$$M_{n+1} \leqslant 2M_n + 1$$

Taup1: Pourquoi avoir utilisé $\leqslant$ et non pas $=$? Et puis, on ne connaît pas M_n.

Mathémator : Vous touchez du doigt deux problèmes essentiels qui vont guider bon nombre de nos raisonnements.
Pour le $\leqslant$, il est essentiel de comprendre que nous avons prouvé que $2M_n + 1$ mouvements étaient **suffisants** mais nous ne savons toujours pas si ces $2M_n + 1$ mouvements sont **nécessaires**. C'est très important de comprendre la différence. Par exemple, pour acheter un chocolat chaud à la cafétéria et l'offrir à son professeur adoré, il est **suffisant** d'avoir 10 euros dans sa poche mais ce n'est pas nécessaire. Inversement, il est **nécessaire** d'avoir au moins 20 centimes mais ce n'est malheureusement pas suffisant.
Revenons à nos disques. Peut-on faire mieux que la solution proposée ?
À un moment donné, il *faut* bien bouger le plus grand disque puisqu'il est sur le mauvais piquet. Il *faut* donc qu'il soit tout seul et il *faut* donc avoir bougé auparavant les n autres disques plus petits. Cela *nécessite au minimum* M_n mouvements. Après cela, on peut bouger le grand disque autant de fois qu'on le désire mais à un moment donné, il *faudra* le placer sur le piquet d'arrivée et déplacer les n autres disques sur le piquet d'arrivée ce qui *nécessite* à nouveau M_n mouvements.
Ainsi, il *faudra au minimum* effectuer $2M_n + 1$ mouvements.
On a donc prouvé que :

$$M_{n+1} \geqslant 2M_n + 1$$

Il est donc **nécessaire** d'effectuer $2M_n + 1$ mouvements pour déplacer les $n+1$ disques. Or nous avons montré auparavant que ces $2M_n + 1$ mouvements étaient **suffisants**. On en déduit donc que :

$$\begin{cases} M_0 = 0 \\ M_n = 2M_{n-1} + 1 \text{ pour tout } n > 0 \end{cases}$$

Est-ce que cette formule marche bien avec nos premiers exemples ?

Taup1: $M_1 = 2 \cdot 0 + 1 = 1$: OK. $M_2 = 2 \cdot 1 + 1 = 3$: OK. $M_3 = 2 \cdot 3 + 1 = 7$: OK.

Mathémator : Ça ne prouve pas grand chose mais ça nous rassure. De toute façon, nous avons prouvé que cette formule est bonne. Que vaut M_{64} ?

Taup1: Ben $M_{64} = 2M_{63} + 1$. Le problème, c'est qu'on ne connaît pas M_{63}.

Mathémator : Et oui : on définit la *suite* **M** à partir d'elle-même, ce qui n'est pas très pratique. On dit qu'on a définit la fonction **récursivement**. Vous verrez cette année en informatique que ce mode de calcul n'est pas réservé aux suites que vous avez peut-être étudiées au lycée. Par exemple, en langage formel, on travaille avec les lettres d'un alphabet. On peut alors définir un mot comme étant soit le vide, soit une lettre suivi d'un mot.

Taup1: C'est un peu idiot : en fait, vous me dites que pour savoir si un truc est un mot, faut déjà savoir si c'est un mot.

Mathémator : Vous n'êtes pas assez précis. Prenons notre alphabet et la chaîne de caractères « math ». C'est le vide suivi de la chaîne « math » qui est la lettre « m » suivie de la chaîne « ath » qui est la lettre « a » suivie de la chaîne « th » qui est la lettre « t » suivie de la lettre « h » qui est un mot puisque c'est une lettre. En remontant, on obtient que « th » est un mot puis que « ath » aussi et enfin « math » est donc un mot.

Taup1 (à part): *Ben on le savait avant. Si c'est ça c'qu'on apprend en info, il est peut-être encore temps de m'inscrire en Khâgne.* **(tout haut)** Fabuleux ! Math est un mot ! J'aurais pas cru, comme ça, a priori.

Mathémator : Ah, comme quoi ce que nous faisons est utile.

Taup1 (à part): *Complètement gâteux le gars* **(tout haut)** Je n'en doute pas.

Mathémator : Et la fin du monde dans tout ça ? Pour connaître M_{64}, il semblerait qu'il faille connaître tous les M_k précédents.
Voyons voir, calculez jusqu'à M_6.

Taup1: Si vous voulez :

- $M_3 = 2 \cdot 3 + 1 = 7$;
- $M_4 = 2 \cdot 7 + 1 = 15$;
- $M_5 = 2 \cdot 15 + 1 = 31$;
- $M_6 = 2 \cdot 31 + 1 = 63$;

Ouais, bof. Je ne vois pas trop où ça nous mène.

Mathémator : « *Ils ont des yeux et ils ne voient pas* ». Et pourtant, quand j'étais jeune, un enfant de six ans aurait reconnu la suite :

$$2^3 - 1,\ 2^4 - 1,\ 2^5 - 1,\ 2^6 - 1, \ldots$$

Taup1 (à part): *Ben tiens ! Les puissances à 6 ans et pourquoi pas les logarithmes à 7.*

Mathémator : Vous dites ?

Taup1: Je réfléchissais.

Mathémator : Ah. Alors, vous avez sûrement envie de dire que tout prête à croire que :

$$M_n = 2^n - 1, \text{ pour tout entier strictement positif } n$$

Taup1: Ben oui, c'est vrai au début donc y a pas de raison pour que ça s'arrête.

Mathémator : Encore faudrait-il le prouver. Par exemple, est-ce que la proposition suivante est un théorème :

Les entiers impairs supérieurs à 3 sont tous des nombres premiers.

Taup1: 3 est premier, 5 est premier, 7 est premier...

Mathémator : ...donc c'est vrai !

Taup1: Ben non. 9 est impair mais n'est pas premier...Ouais, OK, j'ai compris.

Mathémator : « Vrai pour les premiers, vrai pour tous » n'est pas un théorème !
Ici, pour tout entier naturel non nul n, notons $\mathcal{P}_n$ la proposition : $M_n = 2^n - 1$.
Cette proposition est vraie pour $n = 1$ puisque $M_1 = 1 = 2^1 - 1$.
Il existe donc au moins un entier naturel non nul k tel que $\mathcal{P}_k$ soit vraie.
Alors $M_{k+1} = 2M_k + 1 = 2(2^k - 1) + 1 = 2 \cdot 2^k - 2 + 1 = 2^{k+1} - 1$.
Ainsi $\mathcal{P}_k \to \mathcal{P}_{k+1}$ pour tout entier naturel non nul k.
Le tour est joué. On peut donc affirmer que $M_n = 2^n - 1$ pour tout entier naturel non nul n.

Taup1: Donc $M_{64} = 2^{64} - 1$. Je prends ma calculatrice... $M_{64} = 18\,446\,744\,073\,709\,551\,615$. Ça ne m'arrange pas tellement pour connaître la date de la fin du monde.

Mathémator : C'est pourtant simple. Disons que les moines sont très bien entraînés et se relaient efficacement. Ils déplacent alors un disque par seconde. Cela nous donne une durée de jeu de...

Taup1: Ah, ça je sais : $18\,446\,744\,073\,709\,551\,615/3600/24/365,25/100$ ce qui fait en gros 5.8 milliards de siècles...

Mathémator : Et comme l'Univers a grosso modo 140 millions de siècles d'existence, cela nous laisse de la marge.

Taup1: Ouf ! Mais cela ne nous indique pas comment déplacer les huit disques du jeu initial. On sait juste qu'en étant aussi efficaces que les moines, cela nous prendra $M_8 = 2^8 - 1 = 255$ secondes.

Mathémator : Je vous laisse donc cinq minutes pour le faire.

Cinq minutes plus tard

Taup1: Il faut se rendre à l'évidence : je ne suis pas fait pour être moine. Faut le faire pour chaque n jusqu'à 8 en fait et noter comment on a fait. Pfff... C'est pénible.

Mathémator : Et quand un informathicien a bien réfléchi sur le papier, il peut laisser faire le sale boulot à la machine. On a en effet une méthode qu'on sait être correcte. Mais il est très difficile de l'appliquer pour un grand n donné. On sait que ça marche mais on ne sait pas vraiment comment ça marche. Et c'est là qu'une programmation bien pensée devient merveilleusement efficace. Si on a à notre disposition un langage sachant traiter la récursion, il va suffire de lui donner un minimum d'instruction.
Rappelons ici notre méthode : avec $n+1$ disques, je bouge les n plus petits sur le piquet du milieu (M_n mouvements), je bouge le plus grand sur le piquet d'arrivée (1 mouvement) puis je redéplace les n petits sur le grand (M_n mouvements).
Donnez une fonction Python de trois lignes donnant les différents mouvements nécessaires pour résoudre le problème de Hanoï :

Python
```
In [1]: hanoi('A','C','B',2)
Out[1]: '(A -> B)(A -> C)(B -> C)'

In [2]: hanoi('A','C','B',3)
Out[2]: '(A -> C)(A -> B)(C -> B)(A -> C)(B -> A)(B -> C)(A -> C)'
```

Recherche 2 - 9 Rendu de monnaie

Vous diposez de pièces de 1, 2, 5, 10, 50, 100 et 200 centimes d'euro.
Vous devez rendre un certain montant avec ces pièces. Calculez de combien de manières différentes on peut former ce montant avec ces pièces (on en a autant que l'on veut...).

Recherche 2 - 10 Table de Poisson

On veut simuler une v.a.r. X qui suit une loi de Poisson de paramètre $\lambda > 0$ en utilisant une fonction du style `random` qui renvoie une suite aléatoire de nombres selon une distribution uniforme.
Déterminez une fonction Python `poisson(l,n)` qui calcule $\Pr_X(n)$ de manière récursive avec `lambda` le paramètre de la loi. Vous devez obtenir par exemple :

Python
```
In [1]: poisson(1,2)
Out[1]: 0.18393972058572117
```

Fabriquez-vous une fonction `poisson` afin d'obtenir une table :

Python
```
from math import exp
from numpy import array,vectorize

def poisson(l,k):
    ???

def table(deb,fin):
    t =  array([[poisson(j*0.1,i) for j in range(int(deb*10),int(fin*10)-1)] for i in range(8)])
    print(vectorize('%.3f'.__mod__)(t))
```

```
In [2]: table(0.1,1)
[['0.905' '0.819' '0.741' '0.670' '0.607' '0.549' '0.497' '0.449']
 ['0.090' '0.164' '0.222' '0.268' '0.303' '0.329' '0.348' '0.359']
 ['0.005' '0.016' '0.033' '0.054' '0.076' '0.099' '0.122' '0.144']
 ['0.000' '0.001' '0.003' '0.007' '0.013' '0.020' '0.028' '0.038']
 ['0.000' '0.000' '0.000' '0.001' '0.002' '0.003' '0.005' '0.008']
 ['0.000' '0.000' '0.000' '0.000' '0.000' '0.000' '0.001' '0.001']
 ['0.000' '0.000' '0.000' '0.000' '0.000' '0.000' '0.000' '0.000']
 ['0.000' '0.000' '0.000' '0.000' '0.000' '0.000' '0.000' '0.000']]
```

THÈME

3

Complexité

C'est le chapitre œcuménique du programme de deuxième année : l'informatique devient un paradis où s'ébattent dans le plus parfait amour mathématique théorique et sciences expérimentales <3 <3 <3

1 Exemple introductif : problème des 3 sommes

1.1 Méthode scientifique ;-)

Le problème : *on dispose d'une liste de N nombres. Déterminez les triplets dont la somme est nulle.*
Utilisons la force brute (pour le lecteur impatient : nous ferons mieux plus tard...) :

```
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if xs[i] + xs[j] + xs[k] == 0:
                    cpt += 1
    return cpt
```

Comparons les temps pour différentes tailles :

```
In [23]: %timeit trois_sommes([randint(-10000,10000) for k in range(100)])
10 loops, best of 3: 27.5 ms per loop

In [24]: %timeit trois_sommes([randint(-10000,10000) for k in range(200)])
1 loops, best of 3: 216 ms per loop

In [25]: %timeit trois_sommes([randint(-10000,10000) for k in range(400)])
1 loops, best of 3: 1.82 s per loop
```

On peut plutôt s'intéresser aux ratios :

```
from time import perf_counter
from math import log

t = [temps(range(100 * 2**k)) for k in range(4)]

ratio = [t[k + 1] / t[k] for k in range(3)]

def logD(x):
    return log(x)/log(2)

logratio = map(logD,ratio)
```

et on obtient :

```
In [56]: ratio
Out[56]: [7.523860206447286, 9.118789882406599, 8.5098312160934]

In [57]: list(logratio)
Out[57]: [2.940628541715559, 3.133284732580891, 3.128693841844642]
```

Bon, il semble que quand la taille double, le temps est multiplié par 2^3.
Il ne semble donc pas aberrant de considérer que le temps de calcul est de aN^3 mais que vaut a ?

```
In [62]: temps(range(400))
Out[62]: 4.005484320001415
```

$4,00 = a \times 400^3$ donc $a \approx 6,25 \times 10^{-8}$
Donc pour $N = 1000$, on devrait avoir un temps de $6,25 \times 10^{-8} \times 10^9 = 62,5$

```
In [66]: temps(range(1000))
Out[66]: 68.54615448799996
```

Presque mais Python reste Python...
Voici la même chose en C, sans vraiment chercher à optimiser le code.

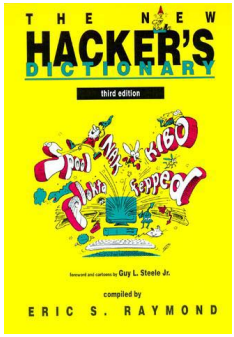
```c
#include <stdio.h>
#include <time.h>

typedef int Liste[13000];

int trois_sommes(int N)
{
  int cpt = 0;
  Liste liste;

  for ( int k = 0; k < N; k++)
    {
      liste[k] = k - (N / 2);
    }

  for (int i = 0; i < N; i++)
    {
      for (int j = i + 1; j < N; j++)
        {
          for (int k = j + 1; k < N; k++)
            {
              if (liste[i] + liste[j] + liste[k] == 0) {cpt++;}
            }
        }
     }
  return cpt;
}

void chrono(int N)
{
  clock_t temps_initial, temps_final;
  float temps_cpu;

  temps_initial = clock();
  trois_sommes(N);
  temps_final   = clock();
  temps_cpu = ((double) temps_final - temps_initial) / CLOCKS_PER_SEC;
  printf("Temps en sec pour %d : %f\n",N, temps_cpu);

}

int main(void)
{
  chrono(100);
  chrono(200);
  chrono(400);
  chrono(800);
  chrono(1600);
  chrono(3200);
  chrono(6400);
  chrono(12800);

  return 1;
}
```

Et on obtient :

```
$ gcc  -std=c99 -Wall -Wextra  -Werror -pedantic -O4 -o somm3 Trois_Sommes.c
$ ./somm3
Temps en sec pour 100 : 0.000000
Temps en sec pour 200 : 0.000000
Temps en sec pour 400 : 0.020000
Temps en sec pour 800 : 0.090000
Temps en sec pour 1600 : 0.720000
Temps en sec pour 3200 : 5.760000
Temps en sec pour 6400 : 45.619999
Temps en sec pour 12800 : 360.839996
```

On a la même évolution en N^3 avec un rapport de 8 entre chaque doublement de taille mais la constante est bien meilleure :
$45,61 = a \times 6400^3$ d'où $a \approx 1,74 \times 10^{-10}$
$360,84 = a \times 12800^3$ d'où $a \approx 1.72 \times 10^{-10}$
Conclusion : Python ou C, l'algo est le même et on constate la même progression selon le cube de la taille.
Ce qui nous intéresse en premier lieu (et ce que l'on mesurera aux concours) est donc un **ORDRE DE CROISSANCE**.
Cependant, les temps sont bien distincts : ici, C va 400 fois plus vite que Python ;-)
On a donc une approche expérimentale et les expériences sont plus facilement menées que dans les autres sciences et on peut en effectuer de très grands nombres.
La mauvaise nouvelle, c'est qu'il est difficile de mesurer certains facteurs comme l'usage du cache, le comportement du ramasse-miettes, etc.
En C, on peut regarder le code assembleur généré mais avec Python, c'est plus mystérieux.
La bonne nouvelle, c'est qu'on peut cependant effectuer une analyse mathématique pour confirmer nos hypothèses expérimentales.

1 2 Notations



Brooks's Law [prov.]

« Adding manpower to a late software project makes it later » – a result of the fact that the expected advantage from splitting work among N programmers is $O(N)$, but the complexity and communications cost associated with coordinating and then merging their work is $O(N^2)$

in « The New Hacker's Dictionnary »
http://outpost9.com/reference/jargon/jargon_17.html#SEC24

Les notations de LANDAU(1877-1938) ont en fait été crées par Paul BACHMANN(1837-1920) en 1894, mais bon, ce sont tous deux des mathématiciens allemands.
Par exemple, si l'on considère l'expression :

$$f(n) = n + 1 + \frac{1}{n} + \frac{75}{n^2} - \frac{765}{n^3} + \frac{\cos(12)}{n^{37}} - \frac{\sqrt{765\,481}}{n^{412}}$$

Quand n est « grand », disons 10 000, alors on obtient :

$$f(10\,000) = 10\,000 + 1 + 0,0001 + 0,00000000075 - 0,0000000000000765 + \text{ peanuts}$$

Tous les termes après n comptent pour du beurre quand n est « grand ». Donnons une définition pour plus de clarté :

Définition 3 - 1

« Grand O »
Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit que f est un « grand O » de g et on note $f = O(g)$ ou $f(n) = O(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \leqslant C|g(n)|$ pour tout $n \in \mathbb{N}$.

Dans l'exemple précédent, $\frac{1}{n} \leqslant \frac{1}{1} \times 1$ pour tout entier n supérieur à 1 donc $\frac{1}{n} = O(1)$.

De même, $\frac{75}{n^2} \leqslant \frac{75}{1^2} \times 1$ donc $\frac{75}{n^2} = O(1)$ mais on peut dire mieux : $\frac{75}{n^2} \leqslant \frac{75}{1} \times \frac{1}{n}$ et ainsi on prouve que $\frac{75}{n^2} = O\left(\frac{1}{n}\right)$.
En fait, un grand O de g est une fonction qui est au maximum majorée par un multiple de g.
On peut cependant faire mieux si l'on a aussi une minoration.
C'est le moment d'introduire une nouvelle définition :

Définition 3 - 2

« Grand Oméga »
Soit f et g deux fonctions de $\mathbb{R}$ dans lui-même. On dit que f est un « grand Oméga » de g et on note $f = \Omega(g)$ ou $f(n) = \Omega(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \geqslant C|g(n)|$ pour tout $n \in \mathbb{N}^*$.

Remarque

Comme Ω est une lettre grecque, on peut, par esprit d'unification, parler de « grand omicron » au lieu de « grand O »...

Remarque

$f = \Omega(g) \iff g = O(f)$...

Si l'on a à la fois une minoration et une majoration, c'est encore plus précis et nous incite à introduire une nouvelle définition :

Définition 3 - 3

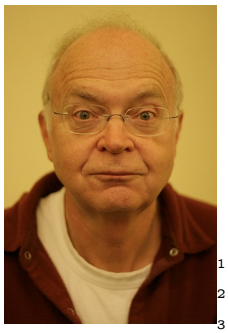
« Grand Théta »
$f = \Theta(g) \iff f = O(g) \quad \wedge \quad f = \Omega(g)$

Voici maintenant une petite table pour illustrer les différentes classes de complexité rencontrées habituellement :

coût \ n	100	1000	10^6	10^9
$\log_2(n)$	≈ 7	≈ 10	≈ 20	≈ 30
$n \log_2(n)$	≈ 665	$\approx 10\,000$	$\approx 2 \cdot 10^7$	$\approx 3 \cdot 10^{10}$
n^2	10^4	10^6	10^{12}	10^{18}
n^3	10^6	10^9	10^{18}	10^{27}
2^n	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$

Gardez en tête que l'âge de l'Univers est environ de 10^{18} secondes...

1 3 Analyse mathématique



D. E. Knuth
Né en 1938

Une bonne lecture de chevet est le troisième volume de « Zi Arte » (Knuth [1998]).
Le temps d'exécution est la somme des produits (coût × fréquence) pour chaque opération.

— le coût dépend de la machine, du compilateur, du langage ;

— la fréquence dépend de l'algorithme, de la donnée en entrée.

Par exemple, l'accès à un élément dans un itérateur en Python n'est pas gratuit.

```
In [12]: xs = range(-500,501)

In [13]: %timeit xs[900]
10000000 loops, best of 3: 129 ns per loop

In [14]: %timeit xs[900] + xs[800]
1000000 loops, best of 3: 292 ns per loop

In [15]: %timeit xs[900] + xs[800] + xs[700]
1000000 loops, best of 3: 424 ns per loop
```

Mais si on travaille sur une liste :

```
In [22]: xs = list(range(-500,501))

In [23]: %timeit xs[900]
10000000 loops, best of 3: 40.7 ns per loop
In [24]: %timeit xs[900] + xs[800]
10000000 loops, best of 3: 114 ns per loop
In [25]: %timeit xs[900] + xs[800] + xs[700]
10000000 loops, best of 3: 165 ns per loop
```

oui mais :

```
In [26]: %timeit range(-500,501)
1000000 loops, best of 3: 292 ns per loop
In [27]: %timeit list(range(-500,501))
100000 loops, best of 3: 13.9 µs per loop
```

Dans notre algorithme des 3 sommes en python, on peut donc gagner du temps :

```
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        xi = xs[i]
        for j in range(i + 1, N):
            sij = xi + xs[j]
            for k in range(j + 1, N):
                if sij + xs[k] == 0:
                    cpt += 1
    return cpt
```

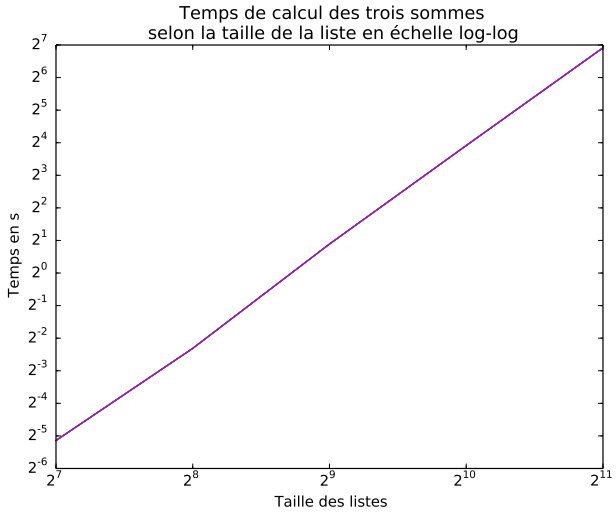
C'est mieux :

```
In [28]:  xs = list(range(-50,51))
In [29]: %timeit trois_sommes(xs)
100 loops, best of 3: 12.9 ms per loop

In [30]:  xs = list(range(-100,101))
In [31]: %timeit trois_sommes(xs)
10 loops, best of 3: 94.4 ms per loop

In [32]:  xs = list(range(-200,201))
In [33]: %timeit trois_sommes(xs)
1 loops, best of 3: 851 ms per loop

In [34]:  xs = list(range(-400,401))
In [35]: %timeit trois_sommes(xs)
1 loops, best of 3: 7.04 s per loop
```

En C, ça ne changera rien, car le compilateur est intelligent et a remarqué tout seul qu'il pouvait garder en mémoire certains résultats.
On peut visualiser en échelle log-log :

```
tailles = [2**k for k in range(7,11)]*
listes = [list(range(- N//2, N//2 + 1)) for N in tailles]
t = [temps(xs) for xs in listes]
for taille in tailles:
    plt.loglog(tailles, t, basex = 2, basey = 2)
```

```
6 plt.title('Temps de calcul selon la taille de la liste en échelle log-log')
7 plt.xlabel('Taille des listes')
8 plt.ylabel('Temps en s')
```



C'est assez rectiligne.

Regardons de nouveau le code des trois sommes et comptons le nombre d'opérations :

OPÉRATION	FRÉQUENCE
Déclaration de la fonction et du paramètre (l. 1)	2
Déclaration de N, cpt et i (l. 2, 3 et 4)	3
Affectation de N, cpt et i (l. 2, 3 et 4)	3
Déclaration de xi (l. 5)	N
Affectation de xi (l. 5)	N
Accès à xs[i] (l. 5)	N
Déclaration de j (l.6)	N
Calcul de l'incrément de i (l. 6)	N
Affectation de j (l.6)	N
Déclaration de sij (l. 7)	S_1
Affectation de sij (l. 7)	S_1
Accès à xs[j] (l.7)	S_1
Somme (l.7)	S_1
Déclaration de k (l.8)	S_1
Incrément de j (l. 8)	S_1
Affectation de k (l.8)	S_1
Accès à x[k] (l.9)	S_2
Calcul de la somme (l.9)	S_2
Comparaison à 0 (l.9)	S_2
Incrément de cpt (l.9)	entre 0 et S_2
Affectation de la valeur de retour (l.11)	1

Que valent S_1 et S_2 ?

$$S_1 = \sum_{i=0}^{N-1} N - (i+1) = \sum_{i'=0}^{N-1} i' = \frac{N(N-1)}{2}$$

$$\begin{aligned}
S_2 &= \sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} N-(j+1)\\
&= \sum_{i=0}^{N-1}\sum_{j'=0}^{N-(i+2)} j'\\
&= \sum_{i=0}^{N-2}\frac{(N-(i+2))(N-(i+1))}{2}\\
&= \sum_{i'=1}^{N-2}\frac{i'(i'+1)}{2}\\
&= \frac{1}{2}\left(\sum_{i=1}^{N-2} i^2+\sum_{i=1}^{N-2} i\right)\\
&= \frac{1}{2}\left(\frac{(N-2)(2N-3)(N-1)}{6}+\frac{(N-2)(N-1)}{2}\right)\\
&= \frac{N(N-1)(N-2)}{6}
\end{aligned}$$

Notons a le temps constant d'affectation, d le temps constant de déclaration, x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.

Le temps d'exécution vérifie :

$$\tau(N)\leqslant(2d+3d+3a+a)+(d+a+x+d+s+a)N+(d+a+x+s+d+s+a)S_1+(x+s+c+s)S_2$$

Or $S_1\sim N^2$ et $S_2\sim N^3$ quand N est « grand ».

Finalement...

$$\tau(N)=O(N^3)$$

...et ce, que l'on utilise C, Python, BrainFuck (https://fr.wikipedia.org/wiki/Brainfuck), etc.

C'est aussi ce que l'on a observé expérimentalement

À retenir

Moralité, on ne s'occupe que de ce qu'il y a dans la boucle la plus profonde et on oublie le reste....

2 Algorithme de Karatsouba

Vous savez additionner deux entiers de n chiffres : cela nécessite $\Theta(n)$ additions de nombres de un chiffre compte-tenu des re-tenues.

Multiplier un entier de n chiffres par un entier de 1 chiffre prend $\Theta(n)$ unités de temps selon le même principe.

En utilisant l'algorithme de l'école primaire pour multiplier deux nombres de n chiffres, on effectue n multiplications d'un nombre de n chiffres par un nombre de 1 chiffre puis une addition des n nombres obtenus. On obtient un temps de calcul en $\Theta(n^2)$.

On peut espérer faire mieux en appliquant la méthode *diviser pour régner*.

On coupe par exemple chaque nombre en deux parties de $m=\lfloor n/2\rfloor$ chiffres :

$$xy=(10^m x_1+x_2)(10^m y_1+y_2)=10^{2m}x_1y_1+10^m(x_2y_1+x_1y_2)+x_2y_2$$

```
Fonction MUL(x:entier ,y: entier):entier
Si n==1 Alors
    Retourner x.y
Sinon
    m ← ⌊n/2⌋
    x1 ← ⌊x/10^m⌋
    x2 ← x mod 10^m
    y1 ← ⌊y/10^m⌋
    y2 ← y mod 10^m
    a ← MUL(x1,y1,m)
    b ← MUL(x2,y1,m)
    c ← MUL(x1,y2,m)
    d ← MUL(x2,y2,m)
    Retourner 10^(2m)a + 10^m(b + c) + d
FinSi
```

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant.

L'addition finale est en $\Theta(n)$ donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \Theta(n) \qquad T(1) = 1$$

Peut-on exprimez $T(n)$ explicitement en fonction de n sans récursion ?

Si nous avions une idée de la solution, nous pourrions la démontrer par récurrence.

Cherchons une idée donc...Ça serait plus simple si n était une puissance de 2. Voyons, posons $n = 2^k$ et $T(n) = T(2^k) = x_k$.

A. Karatsouba (1937 - 2008)

Alors la relation de récurrence devient :

$$x_k = 4x_{k-1} + \Theta(2^k) \qquad x_0 = 1$$

On obtient :

$$x_k = 4(4x_{k-2} + \Theta(2^{k-1})) + \Theta(2^k) = 4^k x_0 + \sum_{i=1}^{k} \Theta(2^k) = 4^k + k\Theta(2^k) = n^2 + \log n \Theta(n) = \Theta(n^2)$$

On montre alors par récurrence forte que ce résultat est vrai pour tout entier naturel non nul n.

Bref, tout ça pour montrer que l'on n'a pas amélioré la situation...

Le grand Андрей Николаевич КОЛМОГОРОВ finit même par conjecturer dans les années 1950 que l'on ne pourra jamais trouver un algorithme de multiplication d'entiers en $o(n^2)$.

Lors d'un séminaire sur ce sujet en 1960, un jeune étudiant soviétique, Анатолий Алексеевич КАРАЦУБА propose cependant au Maître une solution plus simple et navrante de simplicité...

Il fait remarquer à КОЛМОГОРОВ que :

$$bc + ad = ac + bd - (a - b)(c - d)$$

Recherche

En quoi cela simplifie le problème ? Quelle est alors la nouvelle complexité ?

On peut reprendre la méthode précédente ou étudier le paragraphe suivant.

3 Diviser pour régner : ze Master Theorem


Sun Zi (544–496 av. J.-C.)

Le commandement du grand nombre est le même pour le petit nombre, ce n'est qu'une question de division en groupes.

in « L'art de la guerre » 孙子兵法 de Sun Zi (VI^{e} siècle avant JC)

On considère un problème de taille n qu'on découpe en a sous-problèmes de taille n/b avec a et b des entiers. Le coût de l'algorithme est alors :

$$\begin{cases} T(1) = 1 \\ T(n) = a \times T(n/b) + \text{Reconstruction(n)} \end{cases}$$

En général la reconstruction est de l'ordre de $c \times n^{\alpha}$.
Par exemple, pour l'algorithme de КАРАЦУБА nous avions $T(n) = 3T(\lfloor n/2 \rfloor) + \Theta(n)$ donc $a = 3$, $b = 2$, $\alpha = 1$ et c quelconque avec n pair. On obtient :

$$T(n) = aT(n/b) + cn^{\alpha} = a^2T(n/b^2) + ac(n/b)^{\alpha} + cn^{\alpha} = \cdots = a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i c(n/b^i)^{\alpha}$$

On peut poser $n = b^{k_n}$ et alors :

$$T(n) = a^{k_n} + cn^{\alpha} \sum_{i=0}^{k_n-1} (a/b^{\alpha})^i$$

On distingue trois cas :

1. $\boxed{a > b^{\alpha}}$ alors la somme géométrique est équivalente au premier terme négligé à une constante multiplicative et additive près $\left(\frac{1}{a/b^{\alpha}-1}\left(\frac{a}{b^{\alpha}}\right)^{k_n} - \frac{1}{a/b^{\alpha}-1}\right)$ à savoir $(a/b^{\alpha})^{k_n}$ et $T(n) = O(a^{k_n})$ soit $\boxed{T(n) = O(n^{\log_b(a)})}$
2. $\boxed{a = b^{\alpha}}$ alors $\boxed{T(n) = O(n^{\alpha}\log_b(n))}$
3. $\boxed{a < b^{\alpha}}$ alors la série géométrique est convergente : $S(n) \approx n^{\log_b(a)} + cn^{\alpha}/(1 - a/b^{\alpha})$ or $a < b^{\alpha}$ donc $\log_b(a) < \alpha$. Finalement $\boxed{T(n) = O(n^{\alpha})}$

En fait, on peut démontrer (cf Cormen et coll. [2009] pp. 86-97) que les O sont des Θ et considérer des fonctions de reconstruction plus générales..
Pour revenir à l'algorithme de КАРАЦУБА , $a = 3$ et $b^{\alpha} = 2$ donc $T(n) = \Theta(n^{\log_2 3})$.

Master Theorem : totalement hors programme mais qui sait ce qui peut se passer à l'X ou aux ENS...
Soient $a \geqslant 1$ et $b > 1$ deux constantes et $f(n)$ une fonction et T définie par :

$$T(n) = aT(n/b) + f(n)$$

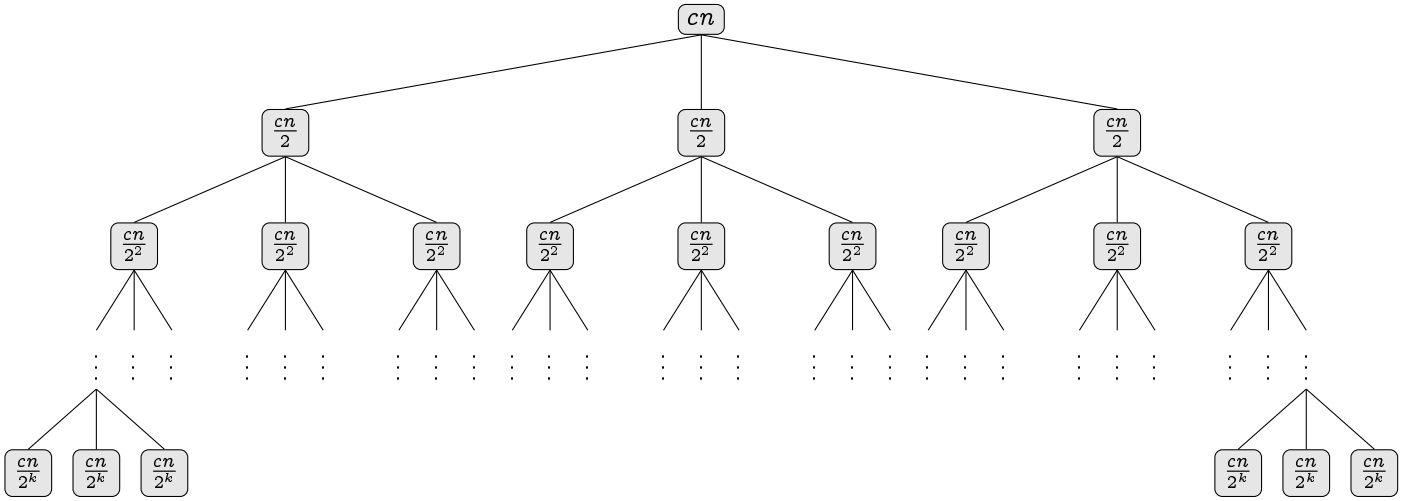
Théorème 3 - 1

avec n/b étant en fait $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$.

1. Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$ alors $T(n) = \Theta(n \log_b a)$
2. Si $f(n) = \Theta(n^{\log_b a})$ alors $T(n) = \Theta(n^{\log_b a} \log_b n)$
3. Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ et si $af(n/b) \leqslant cf(n)$ pour une certaine constante $c < 1$ pour n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Arbre de récursion

Il peut être utile de représenter la récursion par un arbre pour avoir une idée de la formule à démontrer par récurrence. Toujours avec КАРАЦУБА :


cn
$\frac{cn}{2}$
$\frac{cn}{2}$
$\frac{cn}{2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^2}$
$\frac{cn}{2^k}$
$\frac{cn}{2^k}$
$\frac{cn}{2^k}$
$\frac{cn}{2^k}$
$\frac{cn}{2^k}$
$\frac{cn}{2^k}$

RECHERCHES

Recherche 3 - 1

On considère que op est une opération à temps constant.
Quelle est la complexité des boucles suivantes :

```
Pour i variantDe 1 Jusque n Faire
    op
FinPour
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque 90n Faire
        op
    FinPour
    Pour k variantDe 40n Jusque 1 Faire
        op
    FinPour
FinPour
```

```
Pour i variantDe 1 Jusque n*n Faire
    Pour j variantDe 1 Jusque n*3 Faire
        Pour k variantDe 1 Jusque n Faire
            op
        FinPour
    FinPour
FinPour
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque i Faire
        op
    FinPour
FinPour
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque n Faire
        op
    FinPour
    Pour k variantDe 1 Jusque n Faire
        op
    FinPour
FinPour
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque i Faire
        Pour k variantDe 1 Jusque j Faire
            op
        FinPour
    FinPour
FinPour
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque n Faire
        op
    FinPour
FinPour
```

```
s ← 0
i ← n
TantQue i > 0 Faire
    Pour j variantDe 0 Jusque i − 1 Faire
        s ← s + 1
    FinPour
    i ← i // 2
FinTantQue
Retourner s
```

```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe 1 Jusque n Faire
        Pour k variantDe 1 Jusque n Faire
            op
        FinPour
    FinPour
FinPour
```

```
s ← 0
i ← 1
TantQue i < n Faire
    Pour j variantDe 0 Jusque i − 1 Faire
        s ← s + 1
    FinPour
    i ← i * 2
FinTantQue
Retourner s
```

```
s ← 0
i ← 1
TantQue i < n Faire
    Pour j variantDe 0 Jusque n − 1 Faire
        s ← s + 1
    FinPour
    i ← i * 2
FinTantQue
Retourner s
```

Recherche 3 - 2 Point fixe

On dispose d'un tableau de N entiers (relatifs) distincts rangés dans l'ordre croissant.
Déterminez un algorithme qui renvoie l'éventuel indice `i` tel que `a[i] = i`

Recherche 3 - 3 Recherche en dent de scie

Un tableau est constitué d'une suite croissante d'entiers directement suivie d'une suite décroissante d'entiers. Tous les entiers sont supposés distincts. Déterminez une fonction qui détermine si un entier appartient ou non à un tableau en dent de scie. Votre programme doit effectuer $\sim 3\log N$ comparaisons dans le pire des cas.

Recherche 3 - 4 Écart mumixam

Étant donné un tableau de N réels, déterminez un algo en temps linéaire qui détermine la valeur maximum de `a[j] − a[i]` pour $j \geqslant i$.

Recherche 3 - 5 TP de physique

Pour illustrer les lois de la pesanteur, messieurs Alméras et Seigne (en s'inspirant de [Rawlins, 1992]) préparent un TP : ils kidnappent un nombre k d'étudiant(e)s d'ECE et les emmènent à la Tour de Bretagne qui a n étages, le rez-de-chaussée étant l'étage 0.

Le but du TP est de déterminer l'étage à partir duquel lancer un(e) étudiante par la fenêtre lui est fatal.
On suppose qu'un tel étage existe et que le rez-de-chaussée n'est pas fatal..
On supposera que tant que l'étudiant(e) survit, on peut le(a) réutiliser. Pour économiser le temps des expérimentateurs (afin par exemple que M. Alméras continue à corriger vos copies dans la nuit) et réduire l'empreinte carbone de l'expérience (aller-retours en ascenseur), on va chercher un algorithme qui minimise le nombre d'éssais effectués (et non pas le nombre d'étudiant(e)s détruit(e)s :).
Déterminez dans chacun des cas suivants un tel algo :

1. en $O(\log_2(n))$ si $k \geqslant \lceil \log_2(n) \rceil$;
2. en $O(k + n/2^{k-1})$ si $k < \lceil \log_2(n) \rceil$;
3. en $O(\sqrt{(n)})$ si vous ne disposez plus que de 2 étudiant(e)s d'ECE.

Recherche 3 - 6 Exponentiations

On veut calculer x^n avec n un entier naturel et x un élément d'un ensemble muni d'une loi multiplicative associative.
On pose $p_1 = x$ et $p_j = x^j$.

On impose de déterminer un mécanisme qui calcule p_i si on connait déjà $p_1, p_2, \ldots, p_{i-1}$ sous la forme :

$$p_i = p_j \cdot p_k \qquad \text{avec } 0 \leqslant j, k \leqslant i-1$$

1. Donnez un algorithme naïf. Quel est son coût ?
2. On écrit l'exposant en binaire. On remplace chaque 1 par CX et chaque 0 par C. On enlève le premier couple CX (le plus à gauche).

   On traduit C par « mettre au carré » et X par « multiplier par x ».

   Traduisez cet algorithme de manière plus constructive et étudiez sa complexité. Vous donnerez une version impérative et une version récursive.

Recherche 3 - 7 Recherches séquentielles et par saut

Un fichier séquentiel informatisé est schématiquement constitué ainsi, chaque fiche contient

- *L'adresse S de la fiche suivante si elle existe sinon* ***nil*** *pour indiquer la fin du fichier.*
- *L'adresse P de la fiche précédente si elle existe sinon* ***nil*** *pour indiquer qu'il n'y a rien en deçà.*
- *Une donnée D_i alphanumérique*
- *Une première adresse* ***début*** *qui est l'adresse de la première fiche.*

Nous supposerons que le fichier est trié suivant l'ordre croissant et pour simplifier l'étude nous poserons $D_i = i$ (la donnée est égale au numéro de la fiche). Le fichier contient N fiches. Nous appellerons temps d'accès à la fiche numéro i le nombre de fiches lues en partant de *début* pour arriver à la fiche i y compris la fiche i. Ainsi le temps d'accès à la fiche 1 est 1.

Recherche séquentielle.

Pour trouver la fiche i $(1 \leq i \leq N)$ on part de *début*, on lit les fiches dans l'ordre des numéros jusqu'à obtenir la fiche i. Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de *début* à chaque fois, quel est le temps moyen d'accès à une fiche ?

Recherche par saut.

- *Nous supposerons ici que N est le carré d'un entier, $N = a^2$ $(a \in N^*)$. Soit k un entier divisant N et supérieur à 1, $N = kj$.*
- *Nous créons un autre fichier dit de nœuds $N_1, N_2, \ldots, N_j$.*
- *Le départ, noté* **dép**, *contient l'adresse du premier nœud.*
- *Le premier nœud contient l'adresse de la fiche numéro k, la donnée de la fiche numéro k l'adresse du nœud suivant et l'adresse de la fiche $k-1$.*
- *Le deuxième nœud contient l'adresse de la fiche numéro $2k$, la donnée de la fiche numéro $2k$ l'adresse du nœud suivant et de la fiche numéro $2k-1$.*
- ......
- *Le dernier nœud (le $j^{ème}$) contient l'adresse de la fiche numéro N, et celle de $N-1$, la donnée de la fiche numéro N et* ***nil***.

Pour chercher une fiche i on part de **dép**, on passe au premier nœud et on regarde la donnée α portée par le nœud. Si $\alpha < i$ on passe au nœud suivant, sinon on passe à la fiche indiquée par le nœud et, si nécessaire, on revient en arrière dans le fichier jusqu'à atteindre la fiche i. Un temps d'accès à une fiche sera le nombre de nœuds parcourus + le nombre de fiches parcourues y compris celle cherchée.

Si on cherche successivement les fiches 1,2,3,$\cdots$, N en repartant de **dép** à chaque fois, on se propose de déterminer le temps moyen d'accès à une fiche et trouver la valeur de k qui minimise ce temps moyen.

1. Relier la recherche par saut avec la Recherche 3 - 5 page précédente
2. Déterminer le temps d'accès à la fiche 1.
3. Déterminer le temps d'accès à la fiche 2 (si $k > 2$).
4. Déterminer le temps d'accès à la fiche k.
5. Déterminer le temps d'accès à la fiche $k-1$.
6. Déterminer le temps d'accès à la fiche $k+1$.
7. Déterminer le temps d'accès à la fiche ik.
8. Déterminer le temps d'accès à la fiche N.

9. Déterminer le temps d'accès à la fiche j.

10. Déterminer le temps total d'accès aux fiches du $i^{ème}$ paquet.

11. Déterminer le temps total d'accès à toutes les fiches.

12. Déterminer le temps moyen d'accès à une fiche en fonction de k et de N.

13. Déterminer la valeur de k qui minimise ce temps moyen.

14. Comparer le temps moyen d'accès à une fiche par une recherche séquentielle et une recherche par saut en prenant $N = 10\,000$ et une unité de temps égale à $10^{-2}s$.

Recherche 3 - 8 Multiplication du paysan russe

Voici ce qu'apprenaient les petits soviétiques pour multiplier deux entiers. Une variante de cet algorithme a été retrouvée sur le papyrus de Rhind datant de 1650 avant JC, le scribe Ahmes affirmant que cet algorithme était à l'époque vieux de 350 ans. Il a survécu en Europe occidentale jusqu'aux travaux de Fibonacci.

```
1  Fonction MULRUSSE(x:entier ,y: entier, acc:entier):entier
2  Si x==0 Alors
3  |  Retourner acc
4  Sinon
5  |  Si x est pair Alors
6  |  |  Retourner MULRUSSE(x/2,y*2,acc)
7  |  Sinon
8  |  |  Retourner MULRUSSE((x-1)/2,y*2,acc+y)
9  |  FinSi
10 FinSi
```

Que vaut acc au départ ?
Écrivez une version récursive de cet algorithme en évitant l'alternative des lignes 5 à 9.
Écrivez une version impérative de cet algorithme.
Prouvez la correction de cet algorithme.
Étudiez sa complexité.
En python, `x >> 1` décale l'entier x d'un bit vers la doite et `x << 1` décale x d'un bit vers la gauche en complétant par un zéro à droite, `x& y` renvoie l'entier obtenu en faisant la conjonction logique bit à bit des représentations binaires de x et y et `~x` renvoie le complément à 1 de x.
Ré-écrivez la multiplication russe en utilisant que ces opérations bit à bit (pas de division ni de multiplication).

Recherche 3 - 9 Test aléatoire : produit de polynômes correct

On considère deux polynômes écrits différemment, par exemple l'un sous forme factorisée, l'autre sous forme développée : comment faire pour vérifier que les deux polynômes sont bien égaux ?
Quelle est la complexité en nombre de calculs arithmétiques basiques de votre méthode ?
Imaginons maintenant que les polynômes soient de degré d. On choisit un nombre entier n dans l'intervalle $[0, 100d]$ selon une distribution uniforme et on calcule l'image de ce nombre par chacun des polynômes : comment exploiter le résultat ? Évaluer sa « sûreté » ? Complexité ?
Que pensez-vous de l'issue de l'algorithme par rapport aux issues habituelles ? Pouvez-vous prouver que l'algorithme est correct ?
Que se passe-t-il si on répète cette expérience ? Soyez précis(e) dans la description de votre tirage.

Recherche 3 - 10 Produit de matrices correct

On a deux matrices A et B dont on a calculé le produit. On a trouvé C. On voudrait vérifier que le calcul a été bien mené.
Quelle est la complexité habituelle du calcul d'un produit de matrcies ?
On préfère choisir un vecteur $\boldsymbol{v}$ dans $\{0,1\}^n$ et effecteur le produit $D \times \boldsymbol{v}$ en notant $D = A \times B - C$.
D étant non nul, il a au moins un coefficient non nul. On peut se ramener au cas $d_{11} \neq 0$.
Exprimer alors v_1 en fonction des autres v_k et des d_{1k}.
En déduire que si D est non nulle, la probabilité que $D \times \boldsymbol{v} = 0_n$ est inférieure à $1/2$.
Conclure en terme de vérification et de complexité.

Recherche 3 - 11 Algorithme de Strassen

En 1969 (*« Gaussian elimination is not optimal » in Nümerische mathematik*), Konrad Strassen propose un algorithme pour calculer le produit de deux matrices carrées de taille n.
Estimez tout d'abord la complexité en nombre d'additions et en nombre de multiplications de la multiplication usuelle que vous avez vue en Sup.
Considérons à présent le produit de deux matrices de taille 2×2 :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} w & x \\ y & z \end{pmatrix}$$

On note :

$$p_1 = a(f-h),\ p_2 = (a+b)h,\ p_3 = (c+d)e,\ p_4 = d(g-e),\ p_5 = (a+d)(e+h),\ p_6 = (b-d)(g+h),\ p_7 = (c-a)(e+f)$$

alors :

$$\begin{cases} w = p_4 + p_5 + p_6 - p_2 \\ x = p_1 + p_2 \\ y = p_3 + p_4 \\ z = p_1 + p_5 - p_3 + p_7 \end{cases}$$

Comparez le nombre d'opérations effectuées.

Considérez maintenant des matrices carrées de taille 2^t (si ce n'est pas le cas, que peut-on faire ?).

On peut découper nos matrices en quatre blocs de taille 2^{t-1} et appliquer récursivement notre algorithme : il faut quand même vérifier quelque chose pour passer des formules sur les coefficients aux formules sur les matrices : quoi donc ?

Comptez le nombre de multiplications et d'additions nécessaires.

C'est une application de *divide et impera* : pourquoi alors ne pas diviser nos atrices en 9 blocs trois fois plus petits ?

Recherche 3 - 12 Schéma de Horner-Ruffini-Holdred-Newton-Al-Tusi-Liu-Hui...

La méthode que nous allons voir porte le nom du britannique William George Horner (1786 - 1837) mais en fait elle fut publiée presque 10 ans auparavant par un horloger londonien, Theophilius Holdred et simultanément par l'italien Paolo Ruffini (1765 - 1822) mais fut déjà utilisée par Newton 150 ans auparavant et par le chinois Zhu Shije cinq siècles plus tôt (vers 1300) et avant lui par le Persan Sharaf al-Din al-Muzaffar ibn Muhammad ibn al-Muzaffar al-Tusi vers (1100) et avant lui par le Chinois Liu Hui (vers 200) révisant un des résultats présent dans *Les Neuf Chapitres sur l'art mathématique* publié avant la naissance de JC...

Il faut cependant noter que Ruffini l'avait employée en fait comme un moyen de calculer rapidement le quotient et le reste d'un polynôme par $(X-\alpha)$. C'est ce que nous allons (re)découvrir aujourd'hui...

Complexité

Dans toute la suite, un polynôme de dégré n sera représenté par le vecteur de ses coefficients. Par exemple, `[1 2 3]` correspond au polynôme $1+2x+3x^2$.

Prenons l'exemple de $P(x) = 3x^5-2x^4+7x^3+2x^2+5x-3$. Le calcul classique nécessite 5 additions et 15 multiplications.

On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions.
Comparez les complexités au pire du calcul de $P(t)$ pour $t \in \mathbb{K}$. Vous prendrez comme « unité de complexité » les opérations arithmétiques de base : `+ - *`.
Déterminez une fonction `horner(P,t)` qui évalue le polynôme `P` en `t` selon le schéma de HORNER.

Dérivées successives

On note $P_n = a_0 + a_1 X + \cdots + a_n X^n$ un polynôme de degré n.
Déterminez le lien entre l'algorithme de HORNER et la division euclidienne de P_n par $(X - t)$. On utilisera par la suite la notation suivante :

$$P_n = (X - t)(b_n X^{n-1} + b_{n-1} X^{n-2} + \cdots + b_1) + b_0$$

Comment calculer les dérivées successives $\widetilde{P_n^{(j)}}(t)$ en utilisant uniquement des schémas de HORNER ? Quel est l'intérêt informatique ? Voici un peu d'aide...
Considérons le développement de P_n selon les puissances croissantes de $(X - t)$:

$$P_n = t_n (X - t)^n + t_{n-1} (X - t)^{n-1} + \cdots + t_0$$

Montrez que $\widetilde{P_n^{(j)}}(t) = (j!) \times t_j$. Quel est le lien entre les t_j et nos schémas de HORNER ?

Version récursive

Déterminez une fonction récursive `der_j(poly,j,t)` qui calcule $\widetilde{P_n^{(j)}}(t)$ en utilisant le schéma de HORNER.
Pour cela, on pourra commencer par modifier légèrement la fonction `horner` pour récupérer le vecteur $[b_0 b_1 \ldots b_n]$.
Créez ensuite `quotient_horner(poly,t)` qui renvoie le quotient de `poly` par `(X-t)` puis `reste_horner(poly,t)`.
Déduisez-en une fonction récursive `jeme_poly(poly,j,t)` qui renvoie $\widetilde{P_n^{(j)}}(t)$.

Version itérative

Prenons par exemple le polynôme $P_4 = [6, -7, 1, -5, 2]$. Essayez de trouver un algorithme simple pour remplir le tableau suivant :

Restes	0	1	2	3	4	j	
/	6	-7	1	-5	2	a_j	P_4
-12	-9	-1	-1	2	/	b_j	Q_3
						c_j	Q_2
						d_j	Q_1
						e_j	Q_0

En déduire l'écriture de P_4 sous la forme $R_4 + R_3(X-2) + R_2(X-2)^2 + R_1(X-2)^3 + R_0(X-2)^4$ et faire le lien avec les dérivées successives.
Écrivez ensuite une fonction qui renvoie un tableau similaire puis transformez-la un peu pour obtenir les divisions successives en t.

```
jeme_poly_mat([6 -7 1 -5 2],5,2)

   0.     6.  - 7.    1.  - 5.    2.
 - 12.  - 9.  - 1.  - 1.    2.    0.
   1.     5.    3.    2.    0.    0.
   19.    7.    2.    0.    0.    0.
   11.    2.    0.    0.    0.    0.
   2.     0.    0.    0.    0.    0.
```

Recherche 3 - 13 Complexité de l'algorithme d'Euclide et nombre d'or...

On suppose dans toute la suite que a et b sont deux entiers naturels tels que $a \geqslant b$.

1. Rappeler le principe de l'algorithme et de sa preuve.
2. Soit $r_n = a \wedge b$ avec les notations habituelles : $r_0 = b$ puis $r_1 = a \bmod b$, $r_2 = r_1 \bmod r_0$ et plus généralement :

$$r_{k+1} = r_k q_k + r_{k-1}, \quad 0 \leqslant r_{k-1} < r_k$$

   Montrez que pour tout entier $k \in \{1, 2, \ldots, n\}$, on a $q_k \geqslant 1$ puis que $q_n \geqslant 2$.
3. Notons $\Phi = \frac{1+\sqrt{5}}{2}$. Comparez Φ et $1 + \frac{1}{\Phi}$.
4. Montrez par récurrence que $r_k \geqslant \Phi^{n-k}$ pour tout entier $k \in \{0, 1, \ldots, n\}$.
5. Déduisez-en que $b \geqslant \Phi^n$ puis que le nombre d'appels récursifs de l'algorithme d'Euclide est au maximum de $\frac{\ln b}{\ln \Phi}$.

CHAPITRE

4

Tris

Un petit chapitre qui sera le bac à sable de nos expériences sur la complexité...

Dans toute la suite, on se ramènera au cas du tri d'une permutation quelconque de $\mathfrak{S}_n$. On étudiera les danses hongroises disponibles en ligne, par exemple :
https://www.youtube.com/watch?v=kDgvnbUIqT4

1 Tri sélectif...

On parcourt la liste, on cherche le plus grand et on l'échange avec l'élément le plus à droite et on recommence avec la liste privée du plus grand élément.

1.1 Version impérative

On commence par chercher l'indice du maximum d'une liste. On part de 0 et on compare à chaque élément de la liste en faisant évoluer l'indice du maximum si nécessaire.

```
def ind_maxi(xs):
    ind_tmp = 0
    n       = len(xs)
    for i in range(n):
        if xs[i] > xs[ind_tmp]:
            ind_tmp = i
    return ind_tmp
```

Ensuite, on copie la liste donnée en argument pour ne pas l'écraser. On parcourt la liste et on procède aux échanges éventuels entre le maximum et l'élément de droite.

```
def tri_select(xs):
    cs = xs.copy()
    n  = len(cs)
    for i in range(n - 1,0,-1):
        i_m = ind_maxi(cs[:i + 1])
        if i_m != i:
            cs[i_m],cs[i] = cs[i],cs[i_m]
        # print(cs) : pour suivre l'évolution
    return cs
```

On va utiliser `permutation` de la bibliothèque `numpy.random` qui renvoie une permutation uniformément choisie par les permutations de $\mathfrak{S}_n$.

```
In [5]: ls = list(permutation(range(10)))

In [6]: ls
Out[6]: [8, 0, 4, 3, 5, 2, 7, 1, 9, 6]

In [7]: tri_select(ls)
[8, 0, 4, 3, 5, 2, 7, 1, 6, 9]
[6, 0, 4, 3, 5, 2, 7, 1, 8, 9]
[6, 0, 4, 3, 5, 2, 1, 7, 8, 9]
[1, 0, 4, 3, 5, 2, 6, 7, 8, 9]
[1, 0, 4, 3, 2, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 0, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Out[7]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1 2 Correction

Il s'agit de démontrer la correction des deux fonctions.
Pour la première, l'invariant de boucle est : « `ind_temp` est l'indice du maximum des `i + 1` premiers termes de la liste en argument » et se démontre par récurrence.
Pour la deuxième fonction, l'invariant de boucle est « la liste des élements d'indices entre `i` et `n - 1` est triée ».

1 3 Complexité

Nous mesurerons la complexité en nombre de comparaisons.
La complexité de `ind_maxi(xs)` est en $\Theta(n)$ avec n la longueur de `xs`.
En effet, il y a une comparaison par itération et n itérations.
Pour `tri_select`, on effectue $n-1$ itérations, et chacune lance `ind_maxi` sur une liste de longueur $i+1$.
Or $\sum_{i=1}^{n-1} i+1 = \sum_{i=2}^{n} i = (n-2)\frac{2+n}{2} = \frac{n^2-4}{2}$
donc la complexité est en $\Theta(n^2)$.

```
In [10]: ls = list(permutation(100))

In [11]: %timeit tri_select(ls)
1000 loops, best of 3: 561 µs per loop

In [12]: ls = list(permutation(200))

In [13]: %timeit tri_select(ls)
100 loops, best of 3: 2.03 ms per loop

In [14]: ls = list(permutation(400))

In [15]: %timeit tri_select(ls)
100 loops, best of 3: 7.89 ms per loop
```

On observe effectivement que le temps est environ multiplié par 4 quand la taille de la liste double.

2 Tri par insertion

C'est le tri d'un jeu de carte : on insère un élément dans un tableau trié petit à petit en comparant le nouvel élément à insérer aux éléments déjà triés.
La version récursive est assez naturelle. On commence par créer une fonction qui insère un nouvel élément dans la liste :

```
def ins_rec(carte,Main):
    if Main == Vide or carte <= tete(Main):
        return [carte] + Main
    return [tete(Main)] + ins_rec(carte,queue(Main))
```

Ensuite, pour trier, on insère la tête dans la queue triée...

```
def tri_ins_rec(Pioche):
    if Pioche == Vide:
        return Vide
    return ins_rec(tete(Pioche),tri_ins_rec(queue(Pioche)))
```

Pour la version itérative, on module aussi :

```python
def insere(y,xs):
    cs = (xs.copy()) + [y] # xs est triée et on insère y par la doite
    n = len(xs)
    i = n
    while cs[i] < cs[i - 1] and i > 0:
        cs[i - 1], cs[i] = cs[i], cs[i - 1]
        i -= 1
        # print(cs)
    return cs

def tri_insere(Pioche):
    Main = Vide
    for carte in Pioche:
        Main = insere(carte, Main)
        #print(Main)
    return Main
```

La justification et la complexité de ces fonctions seront à travailler en TD, mais la complexité semble encore quadratique :

```
In [29]: ls = list(permutation(100))

In [30]: %timeit tri_insere(ls)
1000 loops, best of 3: 941 µs per loop

In [31]: ls = list(permutation(200))

In [32]: %timeit tri_insere(ls)
100 loops, best of 3: 3.67 ms per loop

In [33]: ls = list(permutation(400))

In [34]: %timeit tri_insere(ls)
100 loops, best of 3: 13.7 ms per loop
```

3 Tri fusion

Cette fois, si le tableau a au plus une valeur, il est trié, sinon on coupe le tableau en deux, on trie ces deux moitiés et on fusionne.

```python
def tri_fusion(xs):
    t = len(xs)
    if t < 2:
        return xs
    return fusion(tri_fusion(xs[:t//2]), tri_fusion(xs[t//2:]))
```

Il reste à définir la fusion :

```python
def fusion(xs,ys):
    if xs == Vide or ys == Vide:
        return xs + ys
    if tete(xs) < tete(ys):
        return [tete(xs)] + fusion(queue(xs),ys)
    return [tete(ys)] + fusion(xs,queue(ys))
```

La version récursive est naturelle mais pose toujours des problèmes en Python : vous chercherez donc une version impérative à titre d'exercice...

L'étude de la terminaison, de la correction et de la complexité devra également être conduite par vos soins.
Au fait :

```
In [46]: ls = list(permutation(100))

In [47]: %timeit tri_fusion(ls)
1000 loops, best of 3: 296 µs per loop

In [48]: ls = list(permutation(200))

In [49]: %timeit tri_fusion(ls)
1000 loops, best of 3: 642 µs per loop

In [50]: ls = list(permutation(1000))

In [51]: %timeit tri_fusion(ls)
100 loops, best of 3: 3.88 ms per loop
```

4 Tri rapide

4.1 Le tri et ses comlexités

Observez et commentez :

```
def partition(pivot,seq):
    p,m,g = [],[],[]
    for item in seq:
        (p if item < pivot else (g if item > pivot  else m)).append(item)
    return p,m,g

def tri_rapide(xs):
    if estVide(xs):
        return Vide
    else:
        pivot = tete(xs)
        p,m,g = partition(pivot, xs)
        return (tri_rapide(p)) + m + (tri_rapide(g))
```

Pour la complexité en moyenne, montrez que

$$K(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(K(i) + K(n-1-i))$$

et en déduire que

$$\frac{K(n)}{n+1} = 2\sum_{i=2}^{n}\frac{i-1}{i(i+1)}$$

Que se passe-t-il dans le pire des cas ?
Dans quel cas est-on sûr que le tri en $n \log n$?

4.2 La médiane en temps linéaire

Il serait très pratique d'avoir une médiane en temps linéaire : nous aurions alors un tri rapide en $n \log n$ dans le pire des cas.
La médiane d'un tableau de n éléments est l'élément de rang $\lfloor \frac{n+1}{2} \rfloor$ comme vous l'avez appris au collège.
L'idée est bien sûr d'éviter de trier le tableau pour trouver la médiane :-)
On cherche donc le tableau de rang $r_m = \lfloor \frac{n+1}{2} \rfloor$: on va utiliser la fonction `partition` en la modifiant légèrement pour qu'elle renvoie l'indice du pivot après partition. Si cet indice est égal

à r_m on a gagné, s'il est plus petit on cherche la médiane sur la partie droite, sinon sur la partie gauche.
Y a plus qu'à...

RECHERCHES

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET DE CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2012

FILIÈRE MP HORS SPÉCIALITÉ INFO
FILIÈRE PC

COMPOSITION D'INFORMATIQUE – B – (XEC)

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Sandwich au jambon

Le problème dit du *sandwich au jambon* ou bien encore appelé théorème de *Stone-Tukey* s'énonce de la manière suivante : un ensemble de n points en dimension d peut toujours être séparé en deux parties de cardinal au plus $\lfloor n/2 \rfloor$ par un hyperplan de dimension $d-1$ (certains points peuvent être dans l'hyperplan), où $\lfloor n/2 \rfloor$ désigne la partie entière de $n/2$. De manière concrète, un ensemble de points dans l'espace peut être séparé en deux parties quasi-égales par un plan. De même un ensemble de points dans le plan peut être séparé en deux par une droite et même en 4 à l'aide de deux droites. Ce sujet porte sur la résolution algorithmique de ce problème et de problèmes connexes selon différentes méthodes.

Dans tout le problème, les tableaux sont indicés à partir de 1. L'accès à la i-ème case d'un tableau *tab* est noté $tab[i]$. Quel que soit le langage utilisé, on suppose que les tableaux peuvent être passés comme arguments des fonctions. En outre, il existe une primitive $\mathsf{allouer}(m, c)$ pour créer un tableau de taille m dont chaque case contient c à l'origine, ainsi qu'une primitive $\mathsf{taille}(t)$ qui renvoie la taille d'un tableau t. Enfin, on disposera d'une fonction `floor`(x) qui renvoie la partie entière $\lfloor x \rfloor$ pour tout réel $x \geq 0$.

La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de P. Lorsque cette complexité dépend d'un paramètre n, on dira que P a une complexité en $\mathcal{O}(f(n))$, s'il existe $K > 0$ tel que la complexité de P est au plus $Kf(n)$, pour tout n. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Partie I. Grand, petit et médian

Dans cette partie, nous supposerons donné un tableau `tab` contenant n nombres réels. Les indices du tableau vont de 1 à n. Nous dénoterons par `tab[a..b]` le tableau pris entre les indices a et b c'est-à-dire les cellules $tab[a], tab[a+1], \ldots, tab[b-1], tab[b]$. Nous supposerons dans l'énoncé que $a \leq b$.

Nous utiliserons le tableau de taille 11 suivant pour nos exemples :

3	2	5	8	1	34	21	6	9	14	8

Question 1 Écrire une fonction `calculeIndiceMaximum`(tab, a, b) qui renvoie l'indice d'une case où se trouve le plus grand réel de $tab[a..b]$. Sur le tableau précédent avec $a = 1$ et $b = 11$, la fonction renverra 6 car la case 6 contient la valeur 34.

Question 2 Écrire une fonction `nombrePlusPetit`(tab, a, b, val) qui renvoie le nombre d'éléments dans le tableau `tab[a..b]` dont la valeur est plus petite ou égale à val. Sur le tableau exemple, pour une valeur de val égale à 5, et $a = 1, b = 11$, la fonction devra renvoyer la valeur 4 car seuls les nombres $3, 2, 5, 1$ sont inférieurs ou égaux à 5.

Nous allons maintenant calculer un médian d'un tableau. Rappelons qu'une valeur médiane m d'un ensemble E de nombres est un élément de E tel que les deux ensembles $E_{<m}$ (les nombres de E strictement plus petits que m) et $E_{>m}$ (les nombres de E strictement plus grands que m) vérifient $|E_{<m}| \leq \lfloor n/2 \rfloor$ et $|E_{>m}| \leq \lfloor n/2 \rfloor$. Notez que le problème du médian est une reformulation de problème dit du *sandwich au jambon* pris en dimension 1. Une méthode naïve consiste donc à parcourir les éléments de l'ensemble et à calculer pour chacun d'eux les valeurs de $|E_{<m}|$ et $|E_{>m}|$ mais il est possible de faire mieux comme nous allons le voir dans la partie suivante.

Partie II. Un tri pour accélérer

Une méthode plus efficace serait de trier le tableau par ordre croissant tout en prenant la cellule du milieu dans le tableau trié. Cette méthode certes rapide requiert $\mathcal{O}(n \ln n)$ opérations. Il existe une méthode optimale en temps linéaire $\mathcal{O}(n)$ pour trouver le médian d'un ensemble de n éléments. Cette partie a pour but d'en proposer une implémentation.

Une fonction annexe nécessaire pour cet algorithme consiste à savoir séparer en deux un ensemble de valeurs. Soit un tableau tab et un réel appelé pivot $p = tab[i]$, il s'agit de réordonner les éléments du tableau en mettant en premier les éléments strictement plus petits que le pivot $tab_{<p}$, puis les éléments égaux au pivot p, et en dernier les éléments strictement plus grands $tab_{>p}$. Sur le tableau exemple, en prenant comme valeur de pivot 8 on obtiendra le tableau résultat suivant :

3, 2, 5, 1, 6	**8, 8**	21, 34, 9, 14

Notez que dans le résultat les nombres plus petits que le pivot $3, 2, 5, 1, 6$ peuvent être dans n'importe quel ordre les uns par rapport aux autres.

Question 3 Écrire une fonction `partition`$(tab, a, b, indicePivot)$ qui prend en paramètre un tableau d'entiers $tab[a..b]$ ainsi qu'un entier $a \leq indicePivot \leq b$. Soit $p = tab[indicePivot]$. La fonction devra réordonner les éléments de $tab[a..b]$ comme expliqué précédemment en prenant comme pivot le nombre p. La fonction retournera le nouvel indice de la case ou se trouve la valeur p.
Dans cette question, on suppose que les modifications effectuées par la fonction sur le tableau tab sont persistantes, même après l'appel de la fonction.

Remarquons que le $\lfloor n/2 \rfloor$-ème élément dans l'ordre croissant d'un tableau de taille n est un élément médian du tableau considéré. Nous allons donc non pas programmer une méthode pour trouver le médian mais plus généralement pour trouver le k-ème élément d'un ensemble. Nous allons utiliser l'algorithme suivant :

On cherche le k-ème élément du tableau $tab[a..b]$.
- Si $k = 1$ et $a = b$ alors renvoyer $tab[a]$
- Sinon, soit $p = tab[a]$. Partitionner le tableau $tab[a..b]$ en utilisant le pivot p en mettant en premier les éléments plus petits que p. Soit i l'indice de p dans le tableau résultant.
  - Si $i - a + 1 > k$ chercher le k-ème élément dans $tab[a..i-1]$ et renvoyer cet élément.
  - Si $i - a + 1 = k$ renvoyer le pivot.
  - Si $i - a + 1 < k$ chercher le $(k - i + a - 1)$-ème élément dans $tab[i+1..b]$ et renvoyer cet élément.

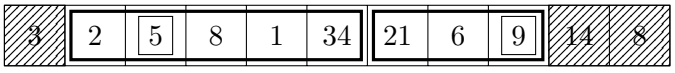
Question 4 Écrire une fonction `elementK`(tab, a, b, k) qui réalise l'algorithme de sélection du k-ème élément dans le tableau $tab[a..b]$ décrit précédemment et renvoie cet élément.

Question 5 Supposons que dans l'algorithme précédent nous voulions rechercher le premier élément mais qu'à chaque étape le pivot choisi est le plus grand élément, quel est un ordre de grandeur du nombre d'opérations réalisées par votre fonction ?

L'algorithme précédent ne semble donc pas améliorer le calcul du médian. Le problème vient du fait que le pivot choisi peut être mauvais c'est-à-dire qu'à chaque étape un seul élément du tableau a été éliminé. En fait, si l'on peut choisir un pivot p dans $tab[a..b]$ tel qu'il y ait au moins $(b-a)/5$ éléments plus petits et $(b-a)/5$ plus grands alors on peut montrer que l'algorithme précédent fonctionne optimalement en temps $\mathcal{O}(n)$.

Pour choisir un tel élément dans $tab[a..b]$, on réalise l'algorithme `choixPivot` suivant où chaque étape sera illustrée en utilisant le tableau donné en introduction en prenant $a = 2$ et $b = 9$.

- On découpe le tableau en paquets de 5 éléments plus éventuellement un paquet plus petit. On calcule l'élément médian de chaque paquet.



S'il n'y a qu'un paquet on renvoie son médian. Sinon on place ces éléments médians au début du tableau.

– On réalise `choixPivot` sur les médians précédents. Dans notre exemple on recommence donc les étapes précédentes en prenant $a = 2$ et $b = 3$.

3	5	9	2	8	1	34	21	6	14	8

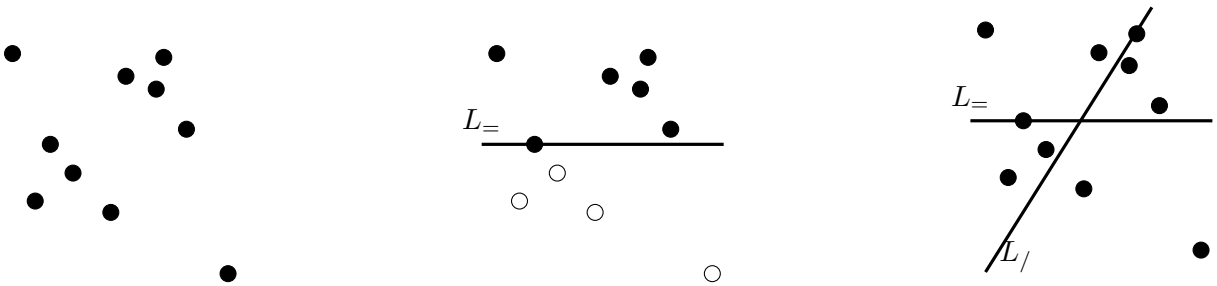
Question 6 Écrire la fonction `choixPivot`(tab, a, b) qui réalise l'algorithme précédent et renvoie la valeur du pivot.

Partie III. De la 1D vers la 2D, des nombres aux points.

Pour un réel $x \geq 0$, on note dans cette partie $\lceil x \rceil$ la partie entière supérieure de x, c'est-à-dire, le plus petit entier qui est plus grand ou égal à x : $\lceil x \rceil - 1 < x \leq \lceil x \rceil$. On supposera disposer d'une fonction `ceil`(x) qui renvoie la partie entière supérieure $\lceil x \rceil$ pour tout réel $x \geq 0$.

Dans la partie précédente, nous avons étudié le problème du médian en dimension 1. On supposera donc que vous disposez maintenant d'une fonction `indiceMedian`(tab, a, b) qui calcule un élément médian du tableau $tab[a..b]$ et renvoie l'indice de cet élément. Vous supposerez de plus que cette fonction ne modifie pas l'ordre des éléments du tableau tab.

Dans cette partie, nous généralisons l'algorithme de manière à trouver deux droites dans le plan séparant un ensemble de n points en 4 parties de cardinal plus petit que $\lceil n/4 \rceil$, Soit E un ensemble de n points tel qu'il n'existe pas trois points alignés. Nous allons chercher deux lignes $L_=$ et $L_/$ séparant cet ensemble de points en 4 parties comme le montre la troisième figure ci-dessous. En effet, dans cette figure chaque partie est composée d'exactement 2 points, les points situés sur les lignes n'étant pas pris en compte. Nous supposerons donnés dans cette partie deux tableaux $tabX$ et $tabY$ de taille n contenant les coordonnées des n points. Ainsi le point i a comme abscisse $tabX[i]$ et comme ordonnée $tabY[i]$.

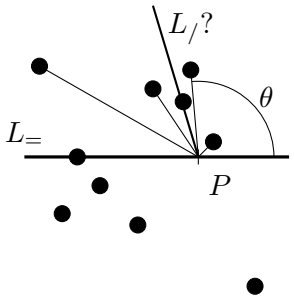


La première étape est de séparer les points en deux ensembles de même cardinal. Il suffit de remarquer que l'on peut toujours effectuer cette séparation par une ligne horizontale notée $L_=$ passant par un point d'ordonnée médiane comme le montre le second schéma ci-dessus.

Question 7 Écrire une fonction `coupeY`$(tabX, tabY)$ qui renvoie l'ordonnée d'une ligne horizontale séparant les points en deux parties de cardinal au plus $\lfloor n/2 \rfloor$.

La seconde ligne $L_/$ est plus difficile à trouver. Nous allons en réalité trouver le point d'intersection des deux lignes $L_=$ et $L_/$.

Soit P un point sur la droite horizontale $L_=$ précédente de coordonnées (x, y). On veut vérifier si ce point peut être le point d'intersection des deux lignes $L_=$ et $L_/$. Nous allons trouver dans un premier temps l'angle entre $L_=$ et $L_/$ en utilisant le fait que $L_/$ doit séparer en deux parties de cardinal proche les points au dessus de $L_=$. Ensuite nous allons vérifier si la droite $L_/$ ainsi devinée sépare les points en dessous de $L_=$ en deux parties presque égales.



Concrètement on considère les demi-droites partant de $P = (x, y)$ et joignant les k points de E au dessus strictement de $L_=$ comme indiqué sur le schéma ci-dessus. On calcule ensuite les angles θ entre $L_=$ et ces demi-droites. Remarquez alors que toute demi-droite d'angle médian partage en deux parties de cardinal $\leq \lfloor k/2 \rfloor$ les points au dessus de $L_=$.

Nous supposerons donnée une fonction `angle`$(x, y, x2, y2)$ qui calcule et renvoie l'angle formé par une demi-droite horizontale partant de (x, y) et allant vers la droite et le segment $(x, y) - (x2, y2)$. La valeur retournée sera comprise dans l'intervalle $[0, 2\pi[$.

Question 8 Écrire une fonction `demiDroiteMedianeSup`$(tabX, tabY, x, y)$ qui calcule et renvoie un angle médian entre $L_=$ et les segments reliant $P = (x, y)$ avec les points de E dont l'ordonnée est supérieure à y.

Pour un point P donné, nous avons déterminé l'angle que doit prendre $L_/$ pour couper les points au dessus de $L_=$ en 2 parties de taille au plus moitié. Il faut maintenant vérifier que cette droite $L_/$ coupe aussi les points en dessous de $L_=$ en 2 parties quasi-égales. Il suffit de vérifier que l'angle de $L_/$ partitionne les angles formés entre P et les ℓ points en dessous de $L_=$ en deux parties de cardinal $\leq \lceil \ell/2 \rceil$.

Question 9 Écrire une fonction `verifieAngleSecondeDroite`$(tabX, tabY, x, y, theta)$ qui calcule les ℓ angles formés entre $L_=$ et les points strictement au dessous de $L_=$. Votre fonction devra renvoyer :

- $\rightarrow$ 0 si theta est une médiane des ℓ angles.
- $\rightarrow$ -1 si le nombre d'angles strictement plus petits que theta est $> \lceil \ell/2 \rceil$
- $\rightarrow$ 1 si le nombre d'angles strictement plus grands que theta est $> \lceil \ell/2 \rceil$

Si $xmin$ est l'abscisse minimale des points de E et $xmax$ l'abscisse maximale, alors il est évident que l'intersection entre $L_=$ et $L_/$ ait une abscisse entre $xmin$ et $xmax$. Nous allons donc rechercher cette abscisse en utilisant l'algorithme suivant :

1. Trouver $L_=$ d'ordonnée y.
2. Soit $\alpha = xmin$ et $\beta = xmax$.
3. On calcule P le point au milieu de (α, y) et (β, y).
4. On calcule l'angle possible de la droite $L_/$ par la fonction `demiDroiteMedianeSup`.
5. Soit d la valeur donnée par la fonction $verifieAngleSecondeDroite$ avec l'angle trouvé précédemment. Si $d = 0$ alors on a trouvé $L_/$. Si $d = -1$ on recommence à partir du point 3 en prenant l'abscisse de P à la place de β. Si $d = 1$ on recommence mais en prenant l'abscisse de P à la place de α.

Question 10 Écrire la fonction `secondeMediane`$(tabX, tabY, y)$ qui à partir d'un ensemble E de points donnés par leurs coordonnées et de l'ordonnée de la droite $L_=$ calcule et renvoie un tableau contenant dans la première case l'abscisse x du point d'intersection de $L_=$ et de $L_/$ ainsi que l'angle de $L_/$ dans la seconde case.

* *
*

Recherche 4 - 1 Stupid Sort

Que pensez-vous de la complexité de cet algorithme de tri ?
(source : http://unclecode.blogspot.tw/2012/02/stupid-sort_2390.html) :

```
void StupidSort()
{
    int i = 0;
    while(i < (size - 1))
    {
           if(data[i] > data[i+1])
          {
                int tmp   = data[i];
                data[i]   = data[i+1];
                data[i+1] = tmp;
                i = 0;
          }
          else
          {
               i++;
          }
     }
}
```

Recherche 4 - 2 Trois-sommes amélioré

Maintenant que vous savez trier et rechercher dichotomiquement, donnez un algo « Trois$_S$ommes ˜en$n^2 \log(n)$ sur des tableaux considérés sans doublons.
(*Cherchez des racines...*)

Recherche 4 - 3 Trois-sommes : variante

On considère trois ensemble d'au maximum N entiers. Déterminez q'il existe un triplet de $A \otimes B \otimes C$ tel que $a+b+c=0$.

Recherche 4 - 4 Mélange de cartes : le mélange de ASF Software

Savoir bien trier c'est aussi savoir bien mélanger.
Afin de convaincre les utilsateurs que leur algorithme de mélange était juste, la société ASF Software, qui produit les logiciels utilisés par de nombreux sites de jeu, avait publié cet algorithme :

```
procedure TDeck.Shuffle;
var
    ctr: Byte;
    tmp: Byte;

    random_number: Byte;
begin
    { Fill the deck with unique cards }
    for ctr := 1 to 52 do
        Card[ctr] := ctr;

    { Generate a new seed based on the system clock }
    randomize;

    { Randomly rearrange each card }
    for ctr := 1 to 52 do begin
        random_number := random(51)+1;
        tmp := card[random_number];
        card[random_number] := card[ctr];
        card[ctr] := tmp;
    end;

```

```
23      CurrentCard := 1;
24      JustShuffled := True;
25  end;
```

1. Considérez un jeu de 3 cartes. Dressez l'arbre de tous les mélanges possibles en suivant cet algorithme. Que remarquez-vous ?
2. Proposez un algorithme qui corrige ce problème. Dressez l'arbre correspondant pour un jeu de trois cartes.
3. Traduisez la fonction proposée en Python puis votre version corrigée en Python.

Recherche 4 - 5 Tri rapide

Explorez les parties non développées en cours sur le tri rapide, notamment la recherche de la médiane.

Recherche 4 - 6 Yet another sort of sort

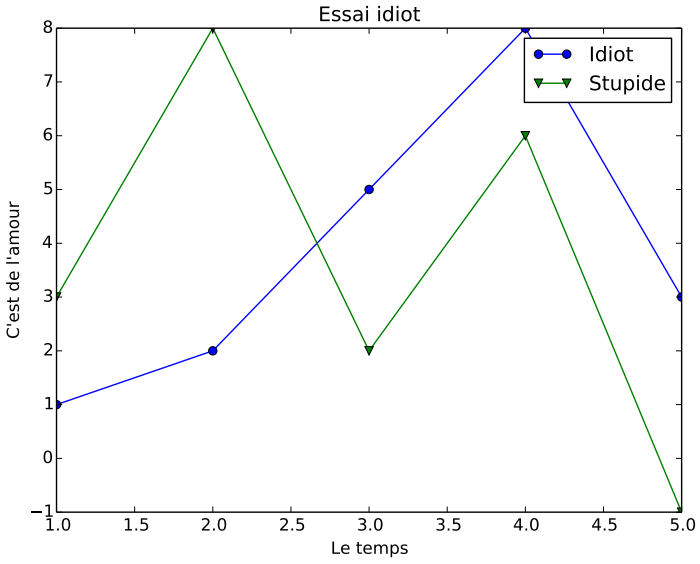
```
Pour i variantDe 1 Jusque n Faire
    Pour j variantDe n  i+1 parPasDe -1 Faire
        Si t[j]<t[j-1] Alors
            Échange t[j] et t[j-1]
        FinSi
    FinPour
FinPour
```

Qu'est-ce que c'est ? Qu'est-ce que ça fait ? Comment ça marche ? Complexité ? En python ? Donnez un nom à ce tri.

Recherche 4 - 7 Comparaisons de tris et révisions sur le tracé de graphiques

Tracez sur un même graphique les temps d'exécution des tris étudiés en fonction de la longueur d'une liste « mélangée ». Sur un autre graphique, comparez ces tris avec des listes ordonnées dans le sens croissant, puis avec des listes ordonnées dans le sens décroissant.
Pour cela, voici quelques rappels sur Matplotlib.
On lance ipython avec l'option `-pylab` ou bien, si vous ne travaillez pas dans un confortable terminal Unix, vous pouvez toujours importer la chose :

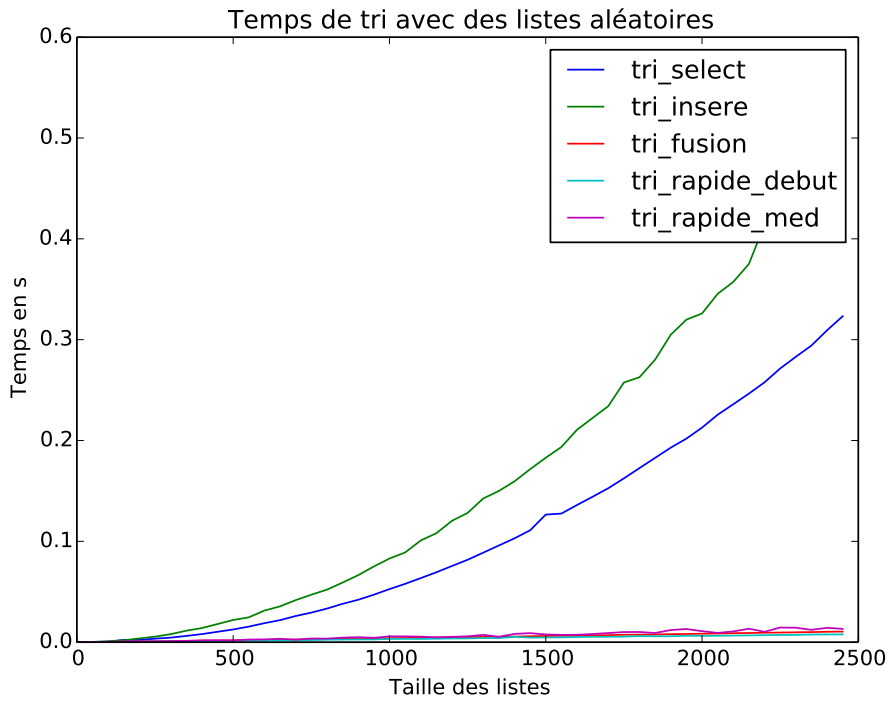
```
In [29]: import matplotlib.pyplot as plt

In [30]: p1 = plt.plot([1,2,3,4,5],[1,2,5,8,3], marker = 'o',label = "Idiot")

In [31]: p2 = plt.plot([1,2,3,4,5],[3,8,2,6,-1], marker = 'v',label = "Stupide")

In [32]: plt.legend()
Out[32]: <matplotlib.legend.Legend at 0x7fbc2c6b76d8>

In [33]: plt.title("Essai idiot")
Out[33]: <matplotlib.text.Text at 0x7fbc2babc668>

In [34]: plt.xlabel("Le temps")
Out[34]: <matplotlib.text.Text at 0x7fbc2c620898>

In [35]: plt.ylabel("C'est de l'amour")
Out[35]: <matplotlib.text.Text at 0x7fbc2baaa2b0>

In [36]: plt.show()

In [37]: plt.savefig("zig.pdf")

In [38]: plt.clf()
```



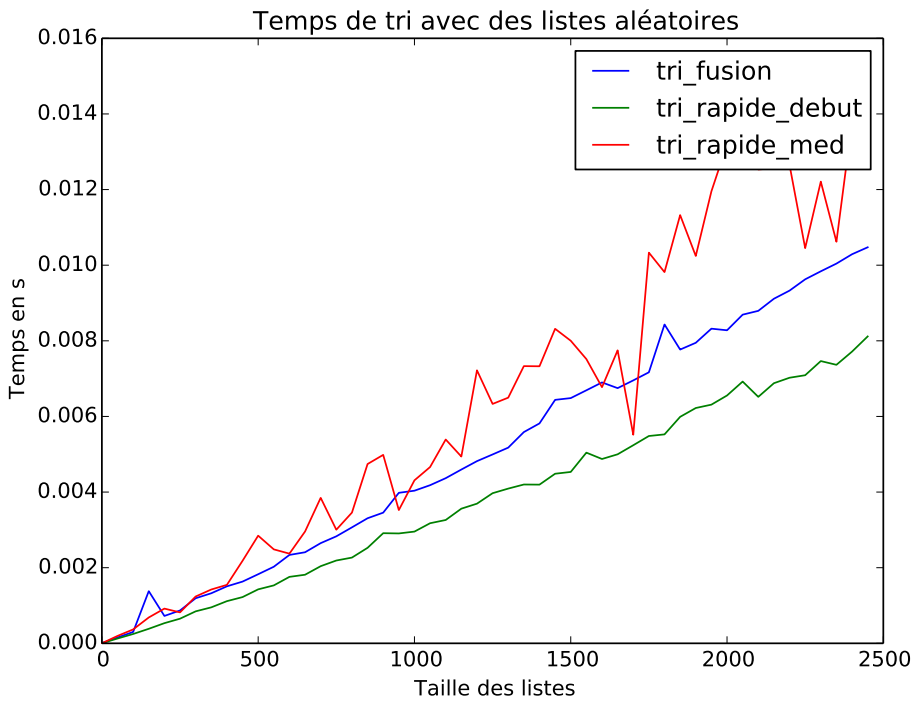
Comment mesurer le temps ? On utilise la fonction `perf_counter` de la bibliothèque `time` :

```
1 from time import perf_counter
2
3 def temps(tri,p):
4     debut = perf_counter()
5     tri(p)
6     return perf_counter() - debut
```

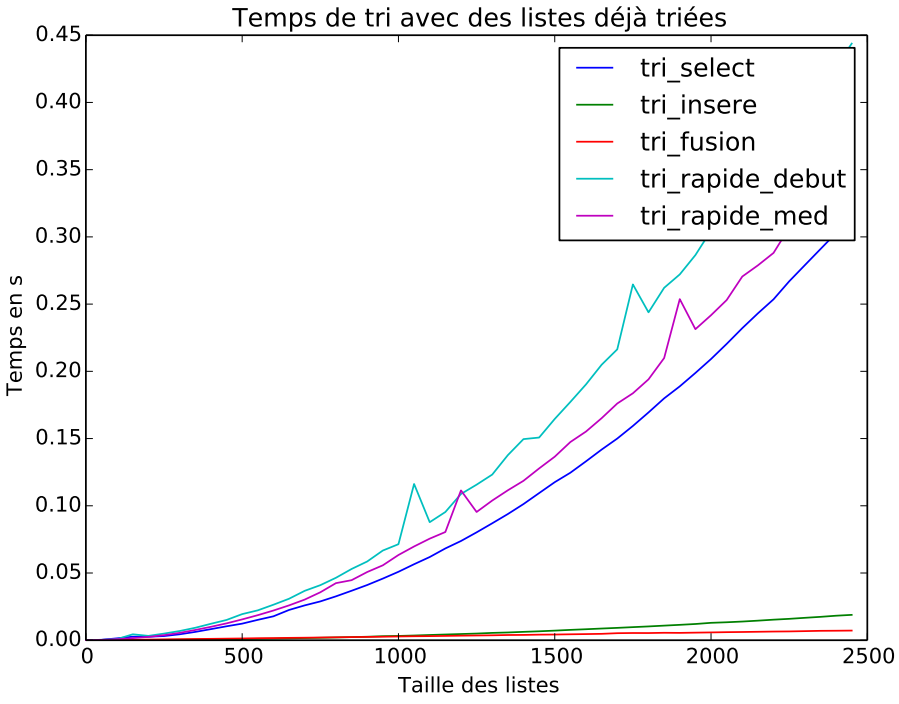
On devrait obtenir :



et en regardant les trois plus rapides :



mais si les listes sont déjà rangées :



Comme le nombre d'appels récursifs va augmenter, on va augmenter aussi le nombre d'appels récursifs autorisé :

```
1 from sys import setrecursionlimit
2 setrecursionlimit(100000)
```

ANNEXE

A

L'aventure géométrique

À la fin des années 1960 au Mit, Seymour Papert, après avoir travaillé avec Jean Piaget, met au point un langage de programmation, LOGO, pour mettre en pratique la thèse qu'il défendra pendant des années : « c'est en créant qu'on apprend ». L'informatique devient alors un outil d'**exploration** de la mathématique. C'est dans cet état d'esprit que nous allons expérimenter des concepts mathématiques très importants : les rapports global/local, implicite/explicite, topologie/-géométrie,...et tout ça avec une petite tortue que nous allons piloter avec Python pour aller effectuer un voyage fantastique au centre de la mathématique.
Pour prolonger vos explorations, n'hésitez pas à dévorer *Turtle geometry* de Harold Abelson et Andrea diSessa paru aux Mit Press en...1981.

Et le taupin créa la tortue

Nous allons créer notre exploratrice mathématique à partir des commandes les plus simples : « tourne », « avance », « saute »...
Voici les principales commandes qui nous permettront de partir en exploration. Analysez-les et comprenez-les et améliorez-les si besoin...

Python

```
from turtle import *

color('red', 'yellow')
speed('fast')

def tourne(angle):
    left(angle)

def avance(longueur):
    forward(longueur)

def saute(longueur):
    penup()
    avance(longueur)
    pendown()

def efface():
    home()
    clear()
```

Expérimentez pour bien comprendre le rôle et les spécificités de chaque procédure.

Polygone

Our Women are Straight Lines.

Our Soldiers and Lowest Class of Workmen are Triangles with two equal sides, each about eleven inches long, and a base or third side so short (often not exceeding half an inch) that they form at their vertices a very sharp and formidable angle. Indeed when their bases are of the most degraded type (not more than the eighth part of an inch in size), they can hardly be distinguished from Straight lines or Women ; so extremely pointed are their vertices. With us, as with you, these Triangles are distinguished from others by being called Isosceles ; and by this name I shall refer to them in the following pages.

Our Middle Class consists of Equilateral or Equal-Sided Triangles.

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order ; and this is the highest class of all.

in « Flatland : A Romance of Many Dimensions » de Edwin Abbott (1884)

Des expériences

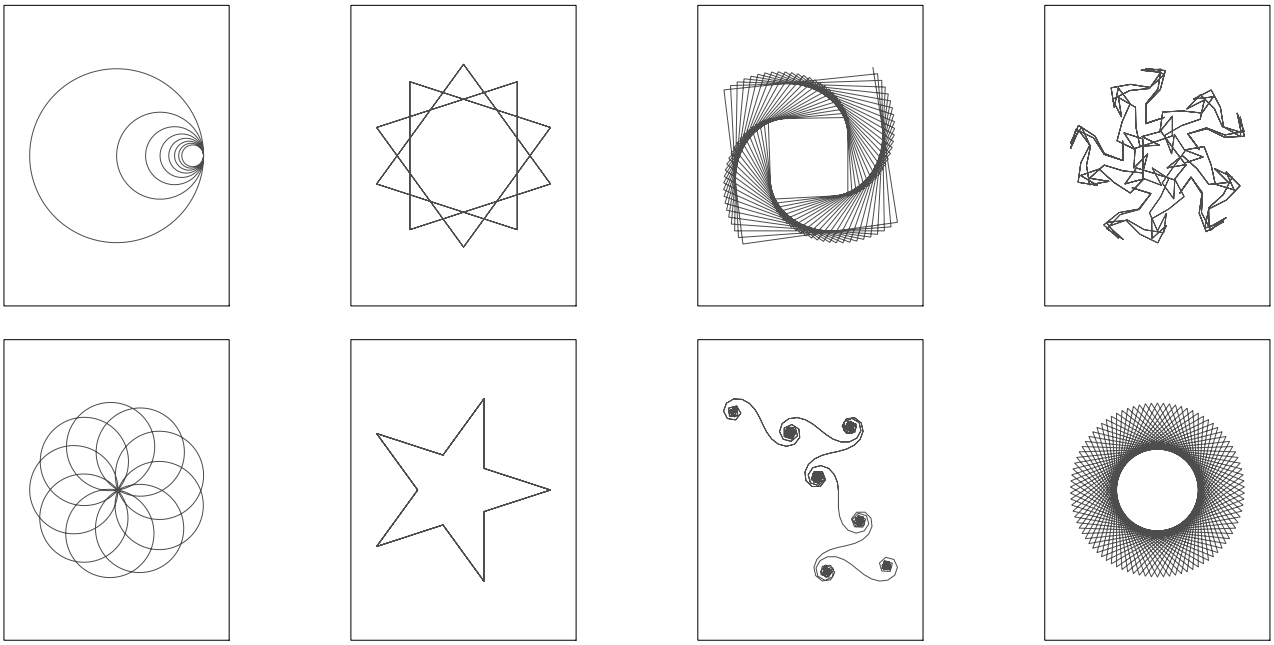
Nous voici prêts : en utilisant uniquement les commandes précédentes, commencer par tracer un triangle équilatéral, un carré , puis une procédure **`triangle(cote)`** qui va tracer un triangle équilatéral de côté **`cote`**. On peut faire de même pour les carrés avec une procédure **`carre(cote)`**.
Bon, euh, c'est en traçant des carrés avec une tortue que vous allez rentrer à Ulm ?...
Arrêtons-nous un instant sur la nature de la géométrie de la tortue en la comparant à la géométrie cartésienne : quelles différences voyez-vous entre les deux ?
La différence est encore plus nette lorsqu'il s'agit de tracer un cercle. En géométrie cartésienne, on utilisera l'équation $x^2+y^2 = R^2$.
Comment tracer un cercle avec la tortue ? Quelles notions mathématiques se cachent derrière ?
Que se passe-t-il avec l'objet décrit par $x^2 + 2y^2 = R^2$?
La tortue ne s'occupe donc que des propriétés intrinsèques des figures.

De plus elle n'a de vision que locale : elle n'a pas de vision globale de l'objet qu'elle explore. Cela peut paraître un inconvénient mais c'est en fait une force qui lui permet d'aborder des notions mathématiques bien plus poussées, notamment dans l'exploration des surfaces.

Enfin la tortue utilise des procédures et échappe au formalisme des équations algébriques ce qui permet d'aller plus loin avec des outils simples.

Bon, passons à la récréation...

Dessinez des petites choses comme ça :



avec des procédures du type :

Python

```
poly1(cote,angle,nb) # la tortue avance nb de cote unités et tourne d'angle degrés
poly2(cote,angle,facteur,nb) # on tourne tantôt d'angle, tantôt d'angle*facteur
poly_spirale(cote,angle,inc,nb) # procédure récursive qui incrémente le côté
poly_spirale2(cote,angle,inc,nb) # procédure récursive qui incrémente l'angle
```

Peut-on contrôler le rayon d'un cercle tracé par la tortue ?

Déterminez une procédure qui dessine un arc de cercle.

Le rayon du cercle est une notion globale qui est malgré tout liée à une quantité locale que la tortue peut donc mesurer.

Un théorème

Observez vos **poly1** : semble-t-il y avoir une relation entre le nombre de côtés et l'angle ?

On peut distinguer deux classes de **poly1** par leur « aspect » : est-ce que cela se retrouve dans la relation précédente ?

Que se passe-t-il pour des chemins clos quelconques ? On dira qu'un chemin clos est simple s'il n'a pas de boucle.

Comment peut-on alors retrouver un vieux théorème sur les angles intérieurs d'un triangle ? Et pour des polygones « sans boucle » quelconques ?

Où se retrouvent les sommets de n'importe quel **poly1** ? Pourriez-vous le démontrer ?

Démontrez alors le théorème suivant :

me A - 1

Un chemin tracé par la procédure **poly1** se refermera exactement lorsque la rotation totale atteindra un multiple de 360 ° .

Écrivez alors une procédure **poly** qui s'arrête quand le chemin se ferme.

Une tortue prédatrice

Notre tortue gagne en confiance et part s'amuser à Las Vegas : comme vous le savez, de nombreuses méthodes de recherche sont plus efficaces si les éléments de l'ensemble exploré sont choisis aléatoirement.

On va tout d'abord observer une tortue se déplaçant aléatoirement : nous allons faire suivre à **avance** et **tourne** une loi uniforme en utilisant la fonction **randint** de Python.

Python
```
from random import randint

In [66]: randint(1,6)
Out[66]: 3
```

Créez alors une fonction :

Python
```
random_move(d1,d2,a1,a2,n)
```

avec d1 et d2 les bornes de la direction, a1 et a2 les bornes de la longueur des pas et n le nombre de pas.

Faites varier les paramètres. Des conjectures ? Comment étudier plus scientifiquement tout ceci ?

Bon, dotons à présent notre tortue d'un odorat et utilisons-la comme chasseuse...

On va dire que l'odeur sera inversement proportionnelle à la distance de la proie. On crée une proie et une fonction qui calcule la distance.

On va dire qu'à chaque étape, la tortue va avancer d'une distance aléatoire et tourner la tête aléatoirement. Si elle sent qu'elle s'éloigne, elle va se tourner d'un angle fixe et recommencer à avancer. Sinon, elle recommence à avancer.

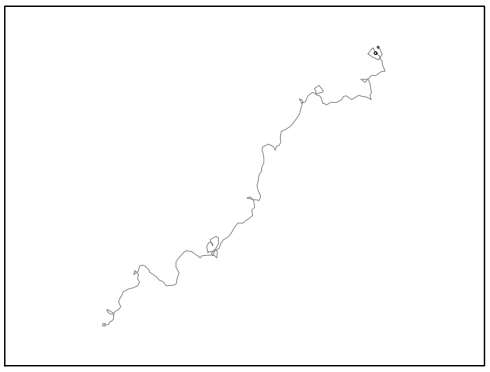
Vous aurez donc une procédure :

Python
```
chasse(d,a,o,n)
```

sachant que la tortue avance d'une distance uniformément choisie dans $\{1,2,..,a\}$, tourne aléatoirement sa tête d'un angle compris entre $-d$ et d pour chercher l'odeur et si elle sent qu'elle s'éloigne, elle tournera toujours la tête de o et ce pendant n étapes.

Par exemple :

Python
```
chasse(60,5,90,200)
```



On peut améliorer le modèle en dotant la tortue de deux narines : selon l'odeur reçue dans chaque narine, elle ira plutôt vers la droite ou la gauche.

On peut ensuite faire bouger la proie qui peut être elle aussi dotée d'un odorat et s'éloignera de sa prédatrice.

Ensuite, il faudra étudier les probas, la dynamique derrière tout ça...tout un sujet de TIPE ;-)

Une tortue topologique

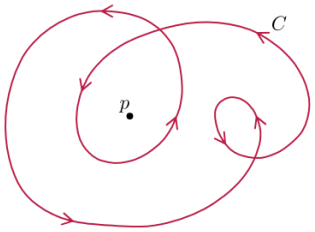
La tortue, en plus de dynamiser la géométrie va aussi la faire fondre. Nous allons en effet travailler à présent sur des objets mous, disons en pâte à modeler.

Dans ce monde, un carré, un triangle, un cercle sont en fait un même objet : quelque chose qui a la même « forme » : pour nous, ce sera le même nombre de « tours » pour parcourir le chemin.

Prenez un triangle, vous le « déformez » petit à petit pour qu'il devienne un carré. Pour le parcourir, la tortue fait toujours un seul tour : on dit alors que le nombre de tours de la tortue (on va l'appeler *indice chélonien* ($\chi\varepsilon\lambda\omega\nu\eta$ voulant dire tortue...)) est un *invariant topologique*.

Lorsque vous effectuez ces « petites » déformations, vous ne pouvez changer l'indice chélonien que faiblement or cet indice est un entier donc il ne va pas bouger.

Bon, c'est un peu intuitif et vasouilleux : peut-on faire mieux sans attendre d'être en Master 2 ?



Il y a des problèmes : on peut faire disparaître une boucle par « petites déformations », on peut en créer trois d'un coup, et en parcourant un cercle dans l'autre sens, on ne déforme rien mais dans tous les cas on change l'indice chélonien tout d'un coup.

Alors c'est quoi une « petite déformation » ? On va dire que c'est une déformation qui ne change pas la procédure de parcours du chemin.

En fait, il faut penser à un chemin comme étant un programme pour tortue (dynamique) et non pas un dessin (statique). . Si vous gardez ça en mémoire quelques années, vous aurez peut-être plus de facilité à aborder la topologie...

Ainsi, ce qui sur le dessin paraît un petit changement en est un énorme dans le programme (il faut faire faire tout d'un coup un tour de plus sur elle-même à la tortue).

Bon, puisqu'il faut regarder les programmes qui font bouger la tortue, il faudrait un peu les normaliser pour les comparer.

Premier problème : `tourne(90)` et `tourne(-270)`. Pour y remédier, on conviendra de n'utiliser que des mesures principales pour les angles (de valeur absolue inférieure à 180).

Ceci nous permet par exemple de pouvoir déceler les petites déformations du dessin qui font disparaître une boucle [a] : **comment ?**

Bon, nous voulons bien croire que deux chemins de même type topologique ont le même coefficient chélonien mais qu'en est-il de la réciproque. Et bien c'est encore vrai et cela constitue un beau théorème publié en 1937 :

Théorème A - 2

Théorème de Whitney-Graustein

Deux chemins fermés du plan peuvent être déformés de l'un vers l'autre si, et seulement si, ils ont le même coefficient chélonien.

Bon, cela n'a pas été écrit comme ça...Vous pouvez lire le court article publié par Whitney à cette adresse :

http://archive.numdam.org/article/CM_1937__4__276_0.pdf

et celui-ci paru vingt ans plus tard sous la plume de John Griffin :

http://archive.numdam.org/article/CM_1956-1958__13__270_0.pdf

Enfin, vous pourrez regarder cette preuve moderne donnée en 2007 par Hansjörg Geiges :

http://arxiv.org/abs/0801.0046

Bon, notre petite tortue nous fait voyager dans une mathématique plutôt moderne...

On pourrait avoir une autre idée « statique » : le nombre de points d'intersection dans le chemin pourrait nous donner aussi un invariant topologique : qu'en pensez-vous ? Quels sont les coefficients chéloniens possibles de chemins ayant un seul point d'intersection ? Deux ? Trois ?

a. Pour les MP* : est-ce que M. Sauvageot vous a déjà fait le coup du nœud de cravate ?

Le tour du monde de la tortue

Une tortue part de l'Équateur, avec un angle de 90 °. Elle arrive au Pôle Nord et tourne à nouveau de 90 °. Elle retourne sur l'Équateur, retourne de 90 ° : elle revient donc à son point de départ. Elle a donc suivi un chemin fermé et elle a tourné de...270 ° ! Voilà qui met à mal notre théorème initial.
De plus, si elle tourne de 70 ° au Pôle Nord, elle aura tourné au total de 250 ° ... Notre coefficient chélonien n'est plus entier ? !
Les déplacements de la tortue sur un plan étaient régis par des **avance**, donc des petits bouts de ligne droite mais au fait, c'est quoi une ligne droite sur une sphère ? Pour que la tortue sache qu'elle va en ligne droite, il faudrait que la caractérisation d'une ligne droite soit locale...
Marcher tout droit, sans tourner, qu'est-ce que c'est ? Essayez de marcher « tout droit » le long d'un parallèle différent de l'Équateur.
Autre problème : la tortue marche « tout droit » d'un Pôle à l'autre (sans utiliser de **tourne**) puis revient au Pôle initial en faisant des pas de côté : elle n'a pas tourné la tête et pourtant que se passe-t-il ?
Maintenant, revenons au grand triangle de 270 °. Imaginons que la tortue garde avec elle un pointeur indiquant sa direction initiale : elle peut faire ça localement avec un peu de mémoire. À chaque fois qu'elle tourne, elle laisse le pointeur vers la direction initiale : que se passe-t-il pour le pointeur quand elle revient au départ ?

ANNEXE

B

POO for dummies

Syntaxe ?

Vous vous êtes sûrement demandé pourquoi les appels de « fonctions » n'ont pas toujours la même syntaxe :

Python

```
In [67]: e = [1,2]

In [68]: len(e)
Out[68]: 2

In [69]: e.len
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-69-8a0fbf71e19e> in <module>()
----> 1 e.len

AttributeError: 'list' object has no attribute 'len'

In [70]: append(e,3)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-70-18248a9bc3f7> in <module>()
----> 1 append(e,3)

NameError: name 'append' is not defined

In [71]: e.append(3)

In [72]: e
Out[72]: [1, 2, 3]
```

Même les nombres ont ce genre de « fonctions » bizarres qui suivent un point :

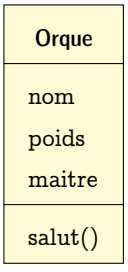
Python

```
In [73]: 1.2.is_integer()
Out[73]: False

In [74]: 1.0.is_integer()
Out[74]: True
```

En fait, cette syntaxe mystérieuse provient du fait que Python est un langage de programmation orientée objet. Mais nous n'avons que quelques minutes pour en parler...

Vocabulaire

Une *classe* correspond à un « moule » permettant de créer un type d'objet.
Il ne faut pas confondre la classe avec l'objet (il vaut mieux manger le gâteau que le moule).
Prenons un exemple : vous êtes Saroumane et vous avez un moule à orques. Vous pouvez créer ainsi une infinité d'orques. Informatiquement la classe `Orque` permet d'*instancier* un objet de type orque.
Un orque a un nom et un poids : ce sont des *attributs* (ses caractéristiques). Il peut effectuer un salut : c'est une *méthode* (ce que l'objet peut faire).
On résume ceci dans un petit tableau :



En python, je pourrais faire ça :

Python

```python
class Orque:

    nom    = ""
    poids  = 0
    maitre = ""

    def salut(self):
        print("Ash nazg durbatulûk! Mon nom est %s" % self.nom)
```

et créer des orques :

Python

```
In [76]: a = Orque()

In [77]: a.nom = "Grishnákh"

In [78]: a.salut()
Ash nazg durbatulûk! Mon nom est Grishnákh

In [79]: a.age = 2

In [80]: b = Orque()

In [81]: b.nom = "Uglúk"

In [82]: b.salut()
Ash nazg durbatulûk! Mon nom est Uglúk

In [83]: a.maitre = "Saroumane"

In [84]: b.maitre
Out[84]: 'Sauron'
```

Les attributs introduits dans la classe ont des valeurs arbitraires et on aurait pu s'en passer comme le montre la création ad hoc de l'attribut age.

Pour clarifier ceci, Python fournit la fonction `__init__()` (notez bien les deux horribles tirets avant et après `init`) qui est appelée à chaque création d'objet et donne une valeur initiale.
On aurait alors pu créer la classe comme ceci :

Python

```python
class Orque:

    def __init__(self):
        self.nom    = ""
        self.poids  = 0
        self.maitre = "Sauron"

    def salut(self):
        print("Ash nazg durbatulûk! Mon nom est %s" % self.nom)
```

On remarque aussi qu'une méthode voulant utiliser un attribut d'une classe doit faire précéder cet attribut du nom de l'instance qui la possède. En python, on utilise le nom standard `self`.

On peut aussi permettre à l'utilisateur(rice) de choisir lui(elle)-même les valeurs par défaut. On ajoute alors des arguments à `__init__` :

Python

```python
class Orque:

    def __init__(self, nom = "", poids = 0, maitre = "Sauron"):
        self.nom    = nom
        self.poids  = poids
        self.maitre = maitre
```

```
    def salut(self):
        print("Ash nazg durbatulûk! Mon nom est %s" % self.nom)
```

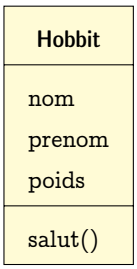
On peut l'utiliser ainsi :

Python

```
In [106]: a = Orque("Truc",3)

In [107]: a.maitre
Out[107]: 'Sauron'

In [108]: a = Orque("Truc","Sauron")

In [109]: a.maitre
Out[109]: 'Sauron'

In [110]: a.poids
Out[110]: 'Sauron'

In [111]: a = Orque("Truc",100,"Sauron")

In [112]: a.poids
Out[112]: 100

In [113]: a.maitre
Out[113]: 'Sauron'

In [114]: a.nom
Out[114]: 'Truc'
```

Héritage

Imaginez maintenant que vous êtes JRR TOLKIEN et que vous voulez fabriquer des Hobbits. Vous vous dites qu'il suffit de créer une classe `Hobbit` :



Python

```
class Hobbit:

    def __init__(self, nom = "", prenom = "", poids = 0):
        self.nom    = nom
        self.prenom = prenom
        self.poids  = poids

    def salut(self):
        print("Bonjour! Mon nom est %s %s" % (self.prenom, self.nom))
```

qui s'utilise comme d'habitude :

Python

```
In [119]: a = Hobbit("Sacquet", "Bilbo", 50)

In [120]: a.salut()
Bonjour! Mon nom est Bilbo Sacquet
```

Et ensuite il y a les Elfes, les Nains, etc.
On remarque cependant que les Orques comme les Hobbits ont un nom et un poids.
On peut alors créer une « super-classe » `Etre` dont `Hobbit` et `Orque` serait des sous-classes qui *héritent* de ses attributs :

Python

```python
class Etre:

    def __init__(self, appellation = "", poids = 0, bonjour = ""):
        self.appellation = appellation
        self.poids       = poids
        self.bonjour     = bonjour

    def salut(self):
        print("%s ! Mon nom est %s" % (self.bonjour, self.appellation))

    def masse(self):
        print("Je pèse %f kg" % self.poids)


class Orque(Etre):

    def __init__(self,nom = "", poids = 0, maitre = "Sauron"):
        Etre.__init__(self,nom,poids,"Ash nazg durbatulûk")
        self.maitre = maitre

class Hobbit(Etre):

    def __init__(self, nom = "", prenom = "", poids = 0):
        Etre.__init__(self,prenom + ' ' + nom, poids, "Bonjour")
        self.prenom  = prenom
```

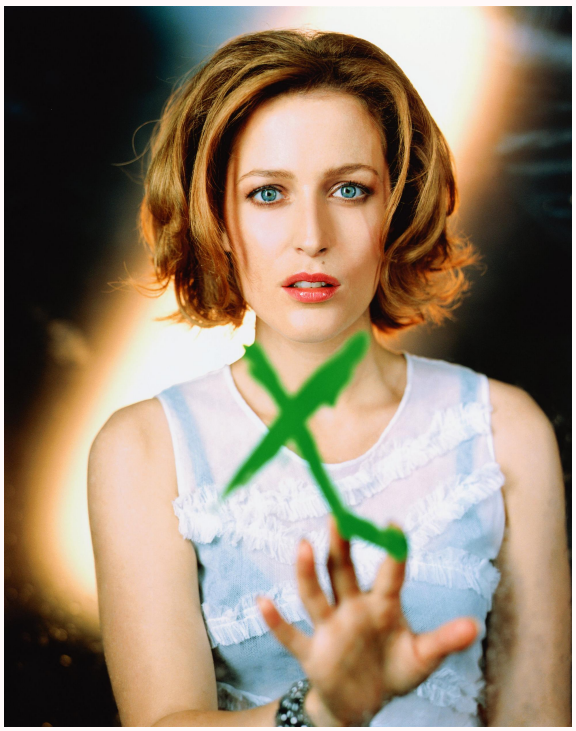
Alors `Hobbit` et `Orque` héritent des attributs et méthodes de `Etre` alors qu'ils ne sont pas définis dans les sous-classes :

Python

```
In [137]: a = Hobbit("Sacquet", "Bilbo", 50)

In [138]: a.masse()
Je pèse 50.000000 kg

In [139]: a.salut()
Bonjour ! Mon nom est Bilbo Sacquet

In [140]: b = Orque("Uglúk",100)

In [141]: b.salut()
Ash nazg durbatulûk ! Mon nom est Uglúk
```

ANNEXE

X C



Armé(e) d'un Python dernière génération, vous allez vous attaquer au sujet X/-Cachan des MP-SI débarqué à Palaiseau ces dernières années...mais vous n'êtes pas seul(e)...

ÉCOLE POLYTECHNIQUE – ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET DE CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2013

FILIÈRE MP HORS SPÉCIALITÉ INFO

FILIÈRE PC

COMPOSITION D'INFORMATIQUE – B – (XEC)

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Points fixes de fonctions à domaine fini

Dans ce problème, on s'intéresse aux points fixes des fonctions $f\colon E \to E$, où E est un ensemble fini. Le calcul effectif et efficace des points fixes de telles fonctions est un problème récurrent en informatique (transformation d'automates, vérification automatique de programmes, algorithmique des graphes, etc), et admet différentes approches selon la structure de E et les propriétés de f.

On suppose par la suite un entier strictement positif $n > 0$ fixé et rangé dans une constante globale de même nom, et on pose $\mathsf{E}_n = \{0, \dots, n-1\}$. On représente une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ par un tableau `t` de taille n, autrement dit $f(x) =$`t[x]` pour tout $x = 0, \dots, n-1$. Ainsi la fonction f_0 qui à $x \in \mathsf{E}_{10}$ associe $2x+1$ modulo 10 est-elle représentée par le tableau

1	3	5	7	9	1	3	5	7	9
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]

.

Les tableaux sont indexés à partir de 0 et la notation `t`$[i]$ est utilisée dans les questions pour désigner l'élément d'indice i du tableau `t`, indépendamment du langage de programmation choisi. Quel que soit le langage utilisé, on suppose qu'il existe une primitive `allouer(n)` pour créer un tableau d'entiers de taille n (le contenu des cases du nouveau tableau est à priori quelconque). On suppose les entiers machines signés, et on suppose que les entiers $-n, -n+1, \dots, n-1, n$ ne débordent pas de la capacité des entiers machines – en d'autres termes, les entiers machines représentent fidèlement ces entiers. On suppose que les tableaux peuvent être passés en argument – le type de passage de paramètre, par valeur ou par adresse, devra être précisé par le candidat si le comportement du code écrit venait à en dépendre. On note dans l'énoncé `vrai` et `faux` les deux valeurs possibles d'un booléen. Le candidat reste libre d'utiliser d'autres notations ou d'autres primitives, pourvu qu'elles existent dans le langage de son choix et qu'elles soient clairement

spécifiées. Enfin, le code écrit devra être sûr (pas d'accès invalide à un tableau, pas de division par zéro, et le programme termine, notamment) pour toutes valeurs des paramètres vérifiant les conditions données dans l'énoncé.

Le temps de calcul d'une procédure `proc` de paramètres $p_1, \dots, p_k$ est défini comme le nombre d'opérations (accès en lecture ou écriture à une case d'un tableau ou à une variable, appel à une des primitives données dans l'énoncé) exécutées par `proc` pour ces paramètres ; on note $T(\texttt{proc}, n)$ le temps de calcul maximal pris sur tous les paramètres possibles pour n fixé. On dit que `proc` s'exécute en temps linéaire si il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha.n \leq T(\texttt{proc}, n) \leq \beta.n$ pour tout $n \geq N$. De même, on dit que `proc` s'exécute en temps logarithmique si il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha \log n \leq T(\texttt{proc}, n) \leq \beta \log n$ pour tout $n \geq N$.

Partie I. Recherche de point fixe : cas général

On rappelle que x est un *point fixe* de la fonction f si et seulement si $f(x) = x$.

Question 1 Écrire une procédure `admet_point_fixe(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si la fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t` admet un point fixe, `faux` sinon. Par exemple, `admet_point_fixe` devra renvoyer `vrai` pour le tableau donné en introduction, puisque 9 est un point fixe de la fonction f_0 qui à x associe $2x + 1$ modulo 10.

Question 2 Écrire une procédure `nb_points_fixes(t)` qui prend en argument un tableau `t` de taille n et renvoie le nombre de points fixes de la fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t`. Par exemple, `nb_points_fixes` devra renvoyer 1 pour le tableau donné en introduction, puisque 9 est le seul point fixe de f_0.

On note f^k l'itérée k-ième de f, autrement dit

$$\begin{array}{rccl} f^k : & \mathsf{E}_n & \to & \mathsf{E}_n \\ & x & \mapsto & \underbrace{f(f(\dots f}_{k \text{ fois}}(x))\dots) \ . \end{array}$$

Question 3 Écrire une procédure `itere(t,x,k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : \mathsf{E}_n \to \mathsf{E}_n$, en deuxième et troisième arguments des entiers x, k de E_n, et renvoie $f^k(x)$.

Question 4 Écrire une procédure `nb_points_fixes_iteres(t,k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : \mathsf{E}_n \to \mathsf{E}_n$, en deuxième argument un entier $k \geq 0$, et renvoie le nombre de points fixes de f^k.

Un élément $z \in \mathsf{E}_n$ est dit *attracteur principal* de $f : \mathsf{E}_n \to \mathsf{E}_n$ si et seulement si z est un point fixe de f, et pour tout $x \in \mathsf{E}_n$, il existe un entier $k \geq 0$ tel que $f^k(x) = z$.

Afin d'illustrer cette notion, on pourra vérifier que la fonction f_1 représentée par le tableau

ci-dessous admet 2 comme attracteur principal.

5	5	2	2	0	2	2
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]

En revanche, on notera que la fonction f_0 donnée en introduction n'admet pas d'attracteur principal, puisque $f_0^k(0) \neq 9$ quel que soit l'entier $k \geq 0$.

Question 5 Écrire une procédure `admet_attracteur_principal(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si et seulement si la fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t` admet un attracteur principal, `faux` sinon. On ne requiert pas ici une solution efficace.

On suppose aux questions 6 et 7 que f admet un attracteur principal. Le *temps de convergence* de f en $x \in \mathsf{E}_n$ est le plus petit entier $k \geq 0$ tel que $f^k(x)$ soit un point fixe de f. Pour la fonction f_1 ci-dessus, le temps de convergence en 4 est égal à 3. En effet, $f_1(4) = 0$, $f_1^2(4) = 5$, $f_1^3(4) = 2$, et 2 est un point fixe de f_1. On note $\mathsf{tc}(f, x)$ le temps de convergence de f en x.

Question 6 Écrire une procédure `temps_de_convergence(t,x)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ qui admet un attracteur principal, en deuxième argument un entier x de E_n, et renvoie le temps de convergence de f en x. On pourra admettre que $\mathsf{tc}(f, x)$ vaut 0 si x est un point fixe de f, et $1 + \mathsf{tc}(f, f(x))$ si x n'est pas un point fixe de f.

Question 7 Écrire une procédure `temps_de_convergence_max(t)` qui prend en argument un tableau `t` de taille n représentant une fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ qui admet un attracteur principal, et renvoie $\max_{i=0\ldots n-1} \mathsf{tc}(f, i)$. On impose un temps de calcul linéaire en la taille n du tableau. À titre d'indication, on pourra au besoin créer un deuxième tableau, qui servira d'intermédiaire au cours du calcul. On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire.

Partie II. Recherche efficace de points fixes

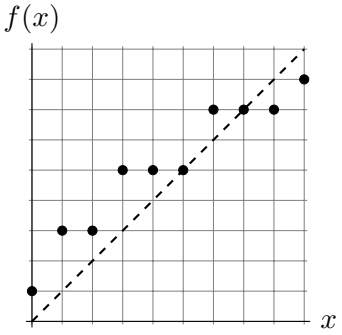
Toute procédure `point_fixe(t)` retournant un point fixe d'une fonction arbitraire est de complexité au mieux linéaire en n. On s'intéresse maintenant à des améliorations possibles de cette complexité lorsque la fonction considérée possède certaines propriétés spécifiques. Nous examinons deux cas.

Premier cas.

Le premier cas que nous considérons est celui d'une fonction croissante de E_n dans E_n. On rappelle qu'une fonction $f : \mathsf{E}_n \to \mathsf{E}_n$ est croissante si et seulement si pour tous $x, y \in \mathsf{E}_n$ tels que $x \leq y$, $f(x) \leq f(y)$.

On admet qu'une fonction croissante de E_n dans E_n admet toujours un point fixe.

À titre d'exemple, la fonction dont le tableau et le graphe sont donnés ci-dessous est croissante. Elle a deux points fixes, à savoir les entiers 5 et 7.



1	3	3	5	5	5	7	7	7	8
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]

Question 8 Écrire une procédure `est_croissante(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si la fonction représentée par `t` est croissante, `faux` sinon. On impose un temps de calcul linéaire en la taille n du tableau. On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire.

Question 9 Écrire une procédure `point_fixe(t)` qui prend en argument un tableau `t` de taille n représentant une fonction croissante $f : \mathsf{E}_n \to \mathsf{E}_n$, et retourne un entier $x \in \mathsf{E}_n$ tel que $f(x) = x$. On impose un temps de calcul logarithmique en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question suivante.

Question 10 Démontrer que la procédure de la question 9 termine. On rappelle que pour prouver qu'une boucle termine, il suffit d'exhiber un entier positif i, fonction des variables du programme, qui décroît strictement à chaque itération de boucle. Justifier que le temps de calcul est logarithmique en la taille n du tableau.

Deuxième cas.

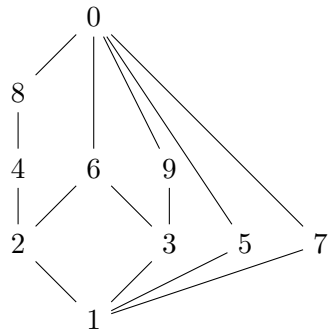
On peut généraliser la notion de fonction croissante comme suit. On rappelle qu'une relation binaire $\preceq$ sur un ensemble E est une relation d'ordre si et seulement si elle est réflexive ($x \preceq x$ pour tout $x \in E$), anti-symétrique (pour tous $x, y \in E$, si $x \preceq y$ et $y \preceq x$, alors $x = y$), et transitive (pour tous $x, y, z \in E$, si $x \preceq y$ et $y \preceq z$, alors $x \preceq z$). Soit $\preceq$ une relation d'ordre sur E. Une fonction $f : E \to E$ est *croissante au sens de* $\preceq$ si et seulement si pour tous $x, y \in E$, $x \preceq y$ implique $f(x) \preceq f(y)$.

Ceci généralise la notion de fonction croissante de E_n dans E_n, que l'on retrouve en prenant $E = \mathsf{E}_n$ et $\preceq$ la relation d'ordre $\leq$. On s'intéresse dorénavant à d'autres relations d'ordre sur E_n.

On dit qu'un élément m de E est un *plus petit élément* de E au sens de $\preceq$ si et seulement si, pour tout $x \in E$, $m \preceq x$. On admet que pour tout ensemble fini E, muni d'une relation d'ordre $\preceq$ et qui admet un plus petit élément m au sens de $\preceq$, pour toute fonction croissante $f : E \to E$ au sens de $\preceq$, il existe un entier $k \geq 0$ tel que $f^k(m)$ est un point fixe de f dans E.

Question 11 Soit E un ensemble fini quelconque muni d'une relation d'ordre $\preceq$ et admettant un plus petit élément m au sens de $\preceq$. Soit $f\colon E \to E$ une fonction croissante au sens de $\preceq$, et soit $k \geq 0$ un entier tel que $f^k(m)$ soit un point fixe de f dans E. Démontrer que $f^k(m)$ est en fait le plus petit point fixe de f au sens de $\preceq$, autrement dit que pour tout autre point fixe x de f dans E, on a $f^k(m) \preceq x$.

Nous nous intéressons maintenant à un choix particulier d'ordre $\preceq$, appelé *ordre de divisibilité* et noté $|$. Précisément, on note $a \mid b$ la relation d'ordre "a divise b" sur les entiers positifs, vraie si et seulement s'il existe un entier $c \geq 0$ tel que $ca = b$. Ainsi, l'ensemble E_{10} ordonné par divisibilité peut se représenter graphiquement comme suit.



D'après la définition donnée précédemment, une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ *croissante au sens de l'ordre de divisibilité* est une fonction telle que pour tous x, y dans E_n, si $x \mid y$, alors $f(x) \mid f(y)$. Par exemple, la fonction représentée par le tableau ci-dessous est croissante au sens de l'ordre de divisibilité.

0	2	4	6	4	8	0	2	0	6
`t[0]`	`t[1]`	`t[2]`	`t[3]`	`t[4]`	`t[5]`	`t[6]`	`t[7]`	`t[8]`	`t[9]`

.

On remarque que, par la question 11, toute fonction de E_n dans E_n croissante au sens de l'ordre de divisibilité a un plus petit point fixe au sens de l'ordre de divisibilité.

On rappelle que le pgcd de deux entiers $x \geq 1$ et $y \geq 1$ est le plus grand entier non nul qui divise x et y. On étend cette définition à des entiers naturels quelconques, en convenant de définir le pgcd d'un entier $x \geq 0$ et de 0 comme valant x.

Question 12 Soit f une fonction de E_n dans E_n, croissante au sens de l'ordre de divisibilité, et notons $x_1, \ldots, x_m$ les points fixes de f dans E_n. Montrer que le plus petit point fixe de f au sens de l'ordre de divisibilité est exactement le pgcd de $x_1, \ldots, x_m$.

Question 13 Écrire une procédure `pgcd_points_fixes(t)` qui prend en argument un tableau `t` de taille n représentant une fonction de E_n dans E_n, croissante au sens de la divisibilité, et renvoie le pgcd de ses points fixes. On impose un temps de calcul logarithmique en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question qui suit.

Question 14 Justifier que la procédure de la question 13 a un temps de calcul logarithmique en la taille n du tableau.

* *
*

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2009

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR FILIÈRE PC

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Chiffrement par blocs

Notation. Dans tout l'énoncé, $[\![a, b[\![$ désigne l'ensemble des entiers naturels supérieurs ou égaux à a et strictement inférieurs à b.

Lorsque l'on souhaite communiquer des données confidentielles, il convient de *chiffrer* ces données, c'est-à-dire de les rendre inintelligibles. Les algorithmes étudiés ici relèvent du chiffrement *symétrique* : une transformation de chiffrement donnée est identifiée par une clé (un entier), qui la désigne et permet également le déchiffrement.

Dans une approche simplifiée du *chiffrement par blocs*, le chiffrement d'un message de taille arbitraire est effectué d'abord en découpant le message en blocs de taille fixée puis en chiffrant chaque bloc. Nous nous limitons ici au chiffrement d'un bloc considéré indépendamment des autres. Dans ce modèle, on se donne un entier $N > 0$, dit *taille* (en pratique N est une puissance de deux). Un bloc (clair ou chiffré) est un entier de $[\![0, N[\![$, et un algorithme de chiffrement est une application de $[\![0, N[\![$ dans $[\![0, N[\![$. Pour permettre le déchiffrement, cette application doit être une permutation de $[\![0, N[\![$ (autrement dit une bijection).

Important. Dans tout le problème, on suppose que le langage de programmation utilisé possède certaines propriétés.

1. Les programmes agissent sur des entiers (naturels) « de taille arbitraire » c'est-à-dire que l'on ignore toutes les questions liées à la taille finie des entiers machine. Autrement dit, on considère que les opérations usuelles ($+$, $*$ etc.) sont celles des entiers naturels.
2. Il existe deux fonctions `rem`(a, b) et `quo`(a, b) calculant respectivement le reste r et le quotient q de la division euclidienne de a par $b > 0$. Il est rappelé que l'égalité $a = bq + r$ et la condition $r < b$ définissent q et r. Autrement dit, si $a = bq + r$, alors il existe un unique quotient q et un unique reste $r < b$, dont les valeurs sont données précisément par les fonctions `quo` et `rem`.

3. Certaines des fonctions demandées sont spécifiées comme renvoyant un tableau ou une liste. Tableau ou liste sont au choix du candidat. En cas de doute, le candidat est invité à définir les primitives dont il juge avoir besoin et à les employer de façon cohérente dans tout le problème.

I. Approche naïve

On cherche à désigner (dans un premier temps) une application arbitraire de $[\![0, N[\![$ (ensemble à N éléments) dans lui-même. Le nombre total de telles applications est N^N.

Considérons un entier k (une clé) pris dans $[\![0, N^N[\![$. L'entier k s'écrit de manière unique sous la forme :

$$k = a_{N-1}N^{N-1} + \cdots + a_i N^i + \cdots + a_1 N^1 + a_0,$$

où chaque coefficient vérifie $a_i \in [\![0, N[\![$ (c'est l'écriture de k en base N). On considère que k représente l'application f_k de $[\![0, N[\![$ dans lui-même définie par $f_k(0) = a_0$, $f_k(1) = a_1$, etc.

Question 1 Écrire la fonction DecomposerBase(N, k) qui prend en arguments la taille N, une clé k de $[\![0, N^N[\![$, et qui renvoie la décomposition de k en base N. En pratique, DecomposerBase renvoie donc le tableau ou la liste des a_i, dans l'ordre des i croissants.

En réalité nous nous intéressons aux permutations de $[\![0, N[\![$. On sait qu'il existe $N!$ permutations d'un ensemble de N éléments. Dans la suite logique de la question précédente, considérons donc une clé k prise dans $[\![0, N![\![$. On admet que k s'écrit de manière unique sous la forme :

$$k = a_{N-1}(N-1)! + a_{N-2}(N-2)! + \cdots + a_i i! + \cdots + a_2 2! + a_1 1! + a_0,$$

où les coefficients vérifient $a_i \in [\![0, i+1[\![$. L'écriture ci-dessus est dite *décomposition sur la base factorielle*. Par exemple, pour $N = 4$ et $k = 17$, on a $k = 2 \cdot 3! + 2 \cdot 2! + 1 \cdot 1! + 0$.

Question 2 Écrire la fonction DecomposerFact(N, k) qui prend en argument la taille N et une clé k de $[\![0, N![\![$, et qui renvoie la décomposition de k sur la base factorielle.

Une fois k décomposée sur la base factorielle, la permutation σ_k de $[\![0, N[\![$ représentée par k se calcule comme suit. En premier lieu, on considère la séquence $\mathcal{L} = (0, 1, \ldots, N-1)$ à N éléments. Cette séquence est modifiée au fur et à mesure que les valeurs prises par la permutation σ_k sont calculées.

La première valeur calculée est $\sigma_k(0)$, égal au $1 + a_{N-1}$-ième élément de $\mathcal{L}$ (c'est-à-dire à a_{N-1}). Une fois $\sigma_k(0)$ calculé, cet entier est retiré de $\mathcal{L}$, qui ne contient plus que $N-1$ entiers.

La seconde valeur calculée est $\sigma_k(1)$, égal au $1 + a_{N-2}$-ième élément de $\mathcal{L}$. Une fois $\sigma_k(1)$ calculé, cet entier est retiré de $\mathcal{L}$. Le procédé est répété jusqu'au calcul de $\sigma_k(N-1)$, égal à l'unique élément de $\mathcal{L}$ restant.

Par exemple, dans le cas $N = 4$, $k = 17$ on a : $\sigma_{17}(0) = 2$ ($a_3 = 2$), et $\mathcal{L}$ devient $(0, 1, 3)$. Ensuite $\sigma_{17}(1) = 3$ ($a_2 = 2$), et $\mathcal{L}$ devient $(0, 1)$. Ensuite $\sigma_{17}(2) = 1$ ($a_1 = 1$), et pour finir $\sigma_{17}(3) = 0$.

Question 3 Écrire la fonction Retirer(L, ℓ, j) qui prend en argument un tableau L à ℓ éléments, et qui renvoie un tableau de taille $\ell - 1$. Le tableau renvoyé est une copie du tableau L dans laquelle le j-ème élément a été retiré.

Question 4 Écrire la fonction EcrirePermutation(N, k) qui prend en arguments la taille N, la clé k de $[\![0, N![\![$, et qui renvoie la permutation σ_k. La permutation sera représentée par le tableau ou la liste des $\sigma_k(i)$, dans l'ordre des i croissants.

Question 5 Écrire les fonctions Chiffrer(N, k, b) et Dechiffrer(N, k, b), qui prennent en arguments la taille N, la clé k et un bloc b . La fonction Chiffrer renvoie $\sigma_k(b)$, tandis que la fonction Dechiffrer renvoie l'unique bloc b' tel que $\sigma_k(b') = b$.

II. Réseau de Feistel

Nous prenons ici le parti de fabriquer des permutations *particulières*. Notre motivation ici est double : (1) réduire la taille des clés (un entier de $[\![0, N![\![$ dans la partie précédente) et (2) effectuer des calculs peu coûteux lors du chiffrement et du déchiffrement.

On commence par fixer la taille à la valeur $N = 2^{64}$. Un bloc b est donc un entier de $[\![0, 2^{64}[\![$. L'ingrédient essentiel du chiffrement est le *réseau de Feistel*. Un réseau de Feistel est une suite de plusieurs opérations, appelées *tours*. Un *tour* est décrit par la figure 1. Sur la figure, *l'entrée* est le bloc $b_i = 2^{32}q_i + r_i$, la *sortie* est $b_{i+1} = 2^{32}q_{i+1} + r_{i+1}$.

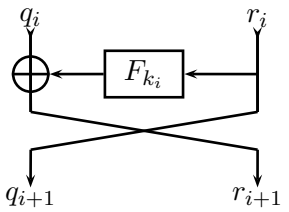


FIG. 1: Un tour de réseau de Feistel

La figure peut aussi se lire comme définisssant q_{i+1} égal à r_i, et r_{i+1} égal à $q_i \oplus F_{k_i}(r_i)$. Le symbole $\oplus$ désigne ici une opération appelée xor. Cette fonction est associative, commutative, et vérifie $\text{xor}(\text{xor}(x, y), y) = x$ pour tout couple d'entiers (x, y). On suppose que la fonction xor est disponible dans le langage de programmation utilisé, accessible sous le nom `xor`. Le symbole F_{k_i} désigne une application sur $[\![0, 2^{32}[\![$, paramétrée par une clé k_i. Par la suite, on suppose donnée une fonction F(k_i, r) qui calcule $F_{k_i}(r)$.

Question 6 Écrire la fonction FeistelTour(k, b) qui prend en argument une clé k et un bloc b (k est un certain k_i, et b est un certain b_i), et renvoie la sortie (notée b_{i+1} ci-dessus) du tour qui utilise la clé k.

Question 7 Écrire la fonction FeistelInverseTour(k, b) qui réalise l'application inverse de la fonction précédente, c'est-à-dire qui calcule et renvoie b_i en fonction de b_{i+1}.

Question 8 Écrire la fonction Feistel(K, ℓ, b) qui prend en entrée le bloc b, et renvoie la sortie d'un réseau de Feistel à ℓ tours. Plus précisément, l'entrée b_0 du premier tour est b, puis l'entrée b_i ($i > 0$) d'un tour est la sortie du tour précédent. Enfin, la sortie du réseau est la sortie b_ℓ du dernier tour. Chaque tour utilise une clé différente. Les clés sont fournies (dans l'ordre) par le tableau K de taille ℓ. Indépendamment du langage de programmation considéré, on supposera qu'un tableau est un argument standard et que ses indices sont les entiers de $[\![0, \ell[\![$.

Question 9 Écrire la fonction FeistelInverse(K, ℓ, b) qui effectue l'opération inverse de la fonction précédente. Cette opération inverse est le déchiffrement, et l'identité suivante doit être vérifiée pour tout bloc b : FeistelInverse$(K, \ell,$ Feistel$(K, \ell, b)) = b$.

III. Vérification de propriétés statistiques

Dans cette partie la taille N est fixée à la valeur $N = 2^{64}$, comme dans la partie précédente. On explore la mise en œuvre de critères de qualité du chiffrement. Certains tests couramment employés sont des tests statistiques effectués sur les message chiffrés. Ces tests servent à mettre en évidence des biais indésirables.

On considère le message clair (infini) formé de la séquence des blocs $0, 1, \ldots$. Pour une permutation de chiffrement des blocs σ, le message chiffré est donc la séquence des blocs $\sigma(0), \sigma(1), \ldots$

Les tests portent sur le message chiffré vu comme une séquence de *bits*, un bit étant un chiffre en base 2, soit 0 ou 1. En fonction d'une longueur paramétrable n, nécessairement multiple de 64, la séquence étudiée est la séquence

$$S_n = \underbrace{\texttt{1010}\cdots\texttt{1101}}_{\sigma(0)\ (64\text{ bits})}\underbrace{\texttt{1001}\cdots\texttt{1110}}_{\sigma(1)\ (64\text{ bits})}\cdots\underbrace{\texttt{1101}\cdots\texttt{0010}}_{\sigma(\frac{n}{64}-1)\ (64\text{ bits})}$$

où par convention, l'écriture binaire (complète) d'un entier x de $[\![0, 2^{64}[\![$, $x = \sum_{i=0}^{63} b_i 2^i$, est la séquence $b_{63}b_{62}\cdots b_1 b_0$ (le bit « le plus significatif » apparaît en premier).

Dans tout ce qui suit, on considère que la permutation étudiée σ est fixée, et calculée par une fonction $\mathsf{Sigma}(x)$, qui prend en entrée un entier x de $[\![0, 2^{64}[\![$ et renvoie un entier de $[\![0, 2^{64}[\![$.

Question 10 Écrire la fonction $\mathsf{Sequence}(n)$ qui construit la séquence S_n ci-dessus, sous la forme d'un tableau de taille n ou d'une liste (on rappelle que n est un multiple de 64). L'ordre des éléments du tableau ou de la liste sera évidemment l'ordre des bits de S_n défini précédemment.

Un premier critère consiste à tester dans quelle mesure les bits `0` et `1` apparaissent avec une fréquence suffisamment proche. Sur un total de n bits ($n \geq 1$), on calcule pour cela la valeur $V_1 = \frac{1}{n}(n_0 - n_1)^2$, où n_0 et n_1 représentent respectivement le nombre de bits `0` et `1` dans la séquence de n bits considérée. En fonction de cette valeur V_1, des tables permettent de dire si un biais statistique est visible.

Question 11 Écrire la fonction $\mathsf{CalculerV1}(n)$ qui détermine la valeur V_1 correspondant à la séquence S_n. Attention, on observera que V_1 n'est pas un entier, il sera représenté en machine par un nombre flottant.

Un second critère généralise le précédent en considérant les séquences de deux bits. Pour n bits ($n \geq 2$), on calcule la valeur V_2 donnée par :

$$V_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1,$$

où $n_{00}, n_{01}, n_{10}, n_{11}$ désignent respectivement le nombre d'occurrences des séquences `00`, `01`, `10`, `11`. On notera qu'on autorise les séquences de deux bits à se recouper. Ainsi la séquence de cinq bits `01100` contient exactement une fois chacune des quatre séquences de deux bits possibles.

Question 12 Écrire la fonction $\mathsf{CalculerV2}(n)$ qui détermine la valeur V_2 correspondant à S_n.

* *
*

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2007

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR FILIÈRE PC

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.
On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction.

⋆ ⋆ ⋆

Compression bzip

Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute :

en temps $O(n^\alpha)$, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn^\alpha$.

Dans ce sujet, il sera question de l'algorithme de Burrows-Wheeler qui compresse très efficacement des données textuelles. Le texte d'entrée à compresser sera représenté par un tableau t contenant des entiers compris entre 0 et 255 inclus.

1 Compression par redondance

La compression par redondance compresse un texte d'entrée qui possède des répétitions consécutives de lettres (ou d'entiers dans notre cas). Dans un premier temps, on calcule les fréquences d'apparition de chaque entier dans le texte d'entrée. Puis on compresse le texte.

Question 1 Écrire la fonction $\mathsf{occurrences}(t, n)$ qui prend en argument un tableau d'entrée t de longueur n ; et qui retourne un tableau r de taille 256 tel que $r[i]$ est le nombre d'occurrences de i dans t pour $0 \leq i < n$.

Question 2 Écrire la fontion $\mathsf{min}(t, n)$ qui prend en argument le tableau t de longueur n ; et qui retourne le plus petit entier de l'intervalle $[0, 255]$ qui apparaît le moins souvent dans le tableau t. (Le nombre d'occurrences de cet entier peut être nul)

L'entier $\mathsf{min}(t, n)$ servira de *marqueur*. On note # pour ce marqueur et, pour simplifier, on suppose que son nombre d'occurrences est nul. Donc $r[\#] = 0$ quand $r = \mathsf{occurrences}(t, n)$. La compression par redondance du texte t fonctionne comme suit : toute répétition maximale contigüe d'une lettre où $t[i] = t[i + 1] = \cdots = t[j] = k$ est codée par les trois entiers $\#, (j - i), k$; toute apparition unique d'une lettre k est codée par cette même lettre.

Par exemple, si le tableau t contient les valeurs $\langle 0,0,3,2,3,3,3,3,3,3,5\rangle$. Le marqueur est donc 1 car 1 n'apparaît pas dans ce tableau. Le texte t' compressé est alors

$$\underbrace{1}_{\#},\underbrace{1,1,0}_{0,0},\underbrace{3}_{3},\underbrace{2}_{2},\underbrace{1,5,3}_{3,3,3,3,3},\underbrace{5}_{5}$$

Question 3 Écrire la fonction tailleCodage(t,n) qui prend comme argument le tableau t et calcule la taille n' du texte compressé ($n'=10$ dans l'exemple ci-dessus).

Question 4 Écrire la fonction codage(t,n) qui prend comme paramètre le tableau t et retourne un tableau d'entiers t' représentant le texte compressé.

Pour pouvoir décoder un texte t' ainsi compressé, il suffit de connaître le marqueur utilisé. Or ce marqueur est le premier entier du texte compressé.

2 Transformation de Burrows-Wheeler

Le codage par redondance n'est efficace que si le texte présente de nombreuses répétitions consécutives de lettres. Ce n'est évidemment pas le cas pour un texte pris au hasard. La transformation de Burrows-Wheeler est une transformation qui, à partir d'un texte donné, produit un autre texte contenant exactement les mêmes lettres mais dans un autre ordre où les répétitions de lettres ont tendance à être contigües. Cette transformation est bijective.

Considérons par exemple le texte d'entrée `concours`. Pour simplifier la présentation, nous utilisons ici des caractères pour le tableau d'entrée. Cependant, dans les programmes, on considère toujours (comme dans la première partie) que le texte d'entrée est un tableau d'entiers compris entre 0 et 255 inclus. Le principe de la transformation suit les trois étapes suivantes :

1 – On regarde toutes les rotations du texte. Dans notre cas, il y en a 8 qui sont :

concours
oncoursc
ncoursco
courscon
oursconc
ursconco
rsconcou
sconcour

2 – On trie ces rotations par ordre lexicographique (l'ordre du dictionnaire).

concour**s**
courscon
ncoursco
oncours**c**
oursconc
rsconcou
sconcou**r**
ursconc**o**

3 – Le texte résultant est formé par toutes les dernières lettres des mots dans l'ordre précédent, soit `snoccuro` dans l'exemple, ainsi que de l'indice de la lettre dans ce texte résultant qui est la première lettre du texte original, soit 3 dans notre exemple. On appelle cet entier la *clé* de la transformation.

On remarque que les deux `c` du texte de départ se retrouvent côte à côte après la transformation. En effet, comme le tri des rotations regroupe les mêmes lettres sur la première colonne, cela conduit à rapprocher aussi les lettres de la dernière colonne qui les précèdent dans le texte d'entrée.

On le constate aussi sur la chaîne : `concours␣de␣l␣ecole␣polytechnique` dont la transformée par Burrows-Wheeler est `sleeeeen␣dlt␣ucn␣ooohcpcc␣iuryqo`.

En pratique, on ne va pas calculer et stocker l'ensemble des rotations du mot d'entrée. On se contente de noter par $rot[i]$ la i-ème rotation du mot. Ainsi, dans l'exemple, $rot[0]$ représente le texte d'entrée `concours`, $rot[1]$ représente `oncoursc`, $rot[2]$ représente `ncoursco`, etc.

Question 5 Écrire la fonction comparerRotations(t, n, i, j) qui prend comme arguments le texte t de longueur n et deux indices i, j ; et qui renvoie, en temps linéaire par rapport à n :

1 si $rot[i]$ est plus grand que $rot[j]$ dans l'ordre lexicographique,

-1 si $rot[i]$ est plus petit que $rot[j]$ dans l'ordre lexicographique,

0 sinon.

On suppose disposer d'une fonction triRotations(t, n) qui trie les rotations du texte donné dans le tableau t en utilisant la fonction comparerRotation. Elle retourne un tableau d'entiers r représentant les numéros des rotations ($rot[r[0]] \leq rot[r[1]] \leq \cdots \leq rot[r[n-1]]$). Cette fonction réalise dans le pire des cas $O(n \ln n)$ appels à la fonction de comparaison.

Question 6 Écrire une fonction codageBW(t, n) qui prend en paramètre le tableau t ; et qui renvoie un tableau contenant le texte après transformation. (La clé sera stockée dans la dernière case de ce tableau)

Question 7 Donner un ordre de grandeur du temps d'exécution de la fonction codageBW en fonction de n.

Pour réaliser l'ensemble du codage, il ne reste plus qu'à réaliser la compression par redondance sur la transformée t' du texte d'entrée t.

3 Transformation de Burrows-Wheeler inverse

Pour décoder le texte t' (`snoccuro3` dans l'exemple) de taille $n' = n + 1$ obtenu après transformation, on construit d'abord un tableau *triCars* de taille n qui contient les mêmes lettres que le texte t' mais dans l'ordre lexicographique croissant. Dans l'exemple, $triCars = \langle c, c, n, o, o, r, s, u \rangle$.

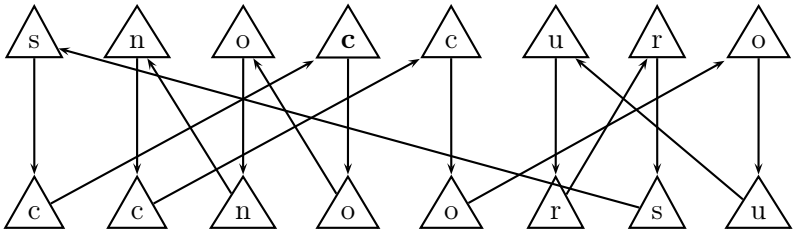
Question 8 Écrire une fonction frequences(t', n') qui prend comme argument un tableau t' de taille n' correspondant au texte codé (avec la clé dans la dernière case) ; et qui renvoie un tableau de taille 256 contenant le nombre d'occurrences de chaque lettre dans t'.

Question 9 Écrire la fonction triCarsDe(t', n') qui part du texte codé t' de taille n' ; et qui renvoie, en temps linéaire par rapport à n, le tableau *triCars* décrit précédemment.

Puis on considère le texte codé t' et le tableau *triCars* précédent (la clé est représentée en gras).

s	n	o	**c**	c	u	r	o	3
c	c	n	o	o	r	s	u	

À chaque lettre de la première ligne, on associe la lettre de la seconde à la même position. À chaque lettre de la deuxième ligne, on associe la même lettre de même rang dans la première ligne. La figure suivante montre ces deux correspondances.



On retrouve le texte de départ `concours` en partant de la clé (position de la lettre en caractère gras) et en suivant les flèches du dessin précédent.

Il faut donc construire le tableau *indices* tel que $indices[i]$ est l'indice de la lettre $triCars[i]$ dans le texte t'. Si plusieurs occurrences de cette lettre figurent dans t', on fait correspondre celle qui figure au même rang dans t'. Le tableau *indices* donne donc la correspondance représentée par les flèches de la seconde ligne vers la première. Sur l'exemple, le tableau *indices* contient les valeurs $\langle 3, 4, 1, 2, 7, 6, 0, 5\rangle$.

Question 10 Écrire la fonction trouverIndices(t', n') prenant en paramètre le texte t' codé de longueur n' ; et qui retourne le tableau *indices* précédemment décrit. Quel est son temps d'exécution en fonction de n' ?

Question 11 Écrire une fonction decodageBW(t', n') qui prend comme paramètre un texte t' de longueur n' ; et retourne le texte t d'origine. Quel est son temps d'exécution en fonction de n' ?

* *
*

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS 2005

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR FILIÈRE PC

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.

Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆⋆⋆

Lignes d'horizon

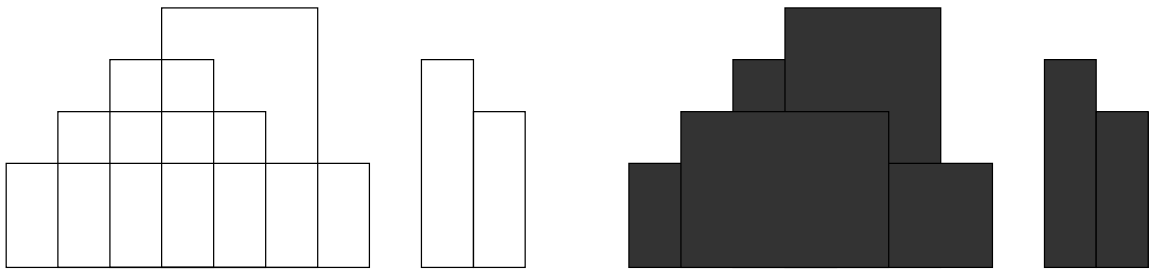
On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. On précisera en tête de copie le langage de programmation utilisé.

Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute :

- en temps linéraire en n, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn$;
- en temps quadratique en n, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn^2$.

I. Détermination de la ligne d'horizon

On cherche à calculer la ligne d'horizon engendrée par n beaux bâtiments modernes ($n > 0$), assimilés à des parallépipèdes verticaux. Pour simplifier, on se place dans l'espace à deux dimensions ; nos bâtiments sont de simples rectangles verticaux ; la ligne d'horizon est représentée par une suite de segments horizontaux, comme indiqué sur la figure suivante :



Dans trois tableaux globaux à valeurs entières g, d et h de longueur n, on range respectivement l'abscisse de gauche, l'abcisse de droite et la hauteur du bâtiment i vérifiant $0 < h[i] < M$ et $0 \leq g[i] < d[i] < N$ pour tout i $(0 \leq i < n)$ où M et N sont deux constantes globales (comme d'habitude, les abcisses croissent de la gauche vers la droite, et les ordonnées du bas vers le haut).

Dans l'exemple ci-dessus, les tableaux g, d et h valent respectivement $\langle 2,3,0,1,8,9\rangle$, $\langle 4,6,7,5,9,10\rangle$ et $\langle 4,5,2,3,4,3\rangle$.

On définit une matrice $E = (e_{i,j})$ $(0 \leq i < M,\ 0 \leq j < N)$ de *pixels* (*picture elements*) dans laquelle on dessine les bâtiments ; chaque élément $e_{i,j}$ vaut 0 ou 1 ; plus exactement $e_{i,j} = 1$ si et seulement si le point de coordonnées réelles $(j+0,5,\ i+0,5)$ est à l'intérieur d'un bâtiment.

Question 1. Écrire une fonction remplir() qui initialise la matrice E.

La ligne d'horizon est représentée par un tableau hor de longueur ℓ contenant une succession d'abcisses et de hauteurs : $hor[2k]$ et $hor[2k+2]$ déterminent les abcisses de début et de fin d'un bout de ligne d'horizon à hauteur $hor[2k+1]$ $(0 \leq k < \ell/2-1)$. Ainsi, dans l'exemple ci-dessus, la ligne d'horizon peut être représentée par le tableau $\langle 0,2,1,3,2,4,3,5,6,2,7,0,8,4,9,3,10,0,N\rangle$ avec $N = 11$. La « sentinelle » N pourra ne pas être le dernier élément du tableau hor ; on autorisera donc les lignes d'horizon contenues dans les seuls premiers éléments d'un tableau.

Dans un premier temps (questions 2 et 3), on se contente d'une ligne d'horizon où tous les segments horizontaux sont de longueur 1. Avec cette convention, on prend, pour représenter la ligne d'horizon correspondant à la figure donnée en exemple, $\langle 0,2,1,3,2,4,3,5,4,5,5,5,6,2,7,0,8,4,9,3,10,0,N\rangle$. Alors la longueur ℓ du tableau vérifie $\ell = 2N+1$ et, pour $0 \leq k \leq N$, $hor[2k] = k$.

Question 2. Écrire une fonction horizon1() qui calcule, à partir de la matrice E, la ligne d'horizon en temps linéaire par rapport au produit MN.

On peut réduire le temps d'exécution de cette fonction en écrivant une fonction qui longe la ligne d'horizon dans la matrice E. Le contour de l'horizon est la ligne continue formée d'une succession de lignes horizontales et de verticales qui borde l'ensemble supérieur des bâtiments. La longueur de ce contour est la somme des longueurs des segments verticaux et horizontaux qui le composent.

Question 3. Écrire une fonction horizon2() qui calcule, à partir de la matrice E, la ligne d'horizon en temps linéaire par rapport à la longueur L du contour de l'horizon.

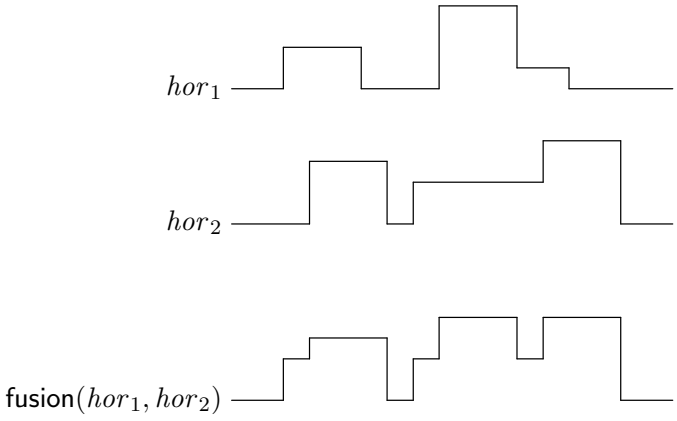
II. Transformations de lignes d'horizon

Un tableau hor, de longueur ℓ, représentant une ligne d'horizon, est en forme canonique si, pour tout k, elle vérifie

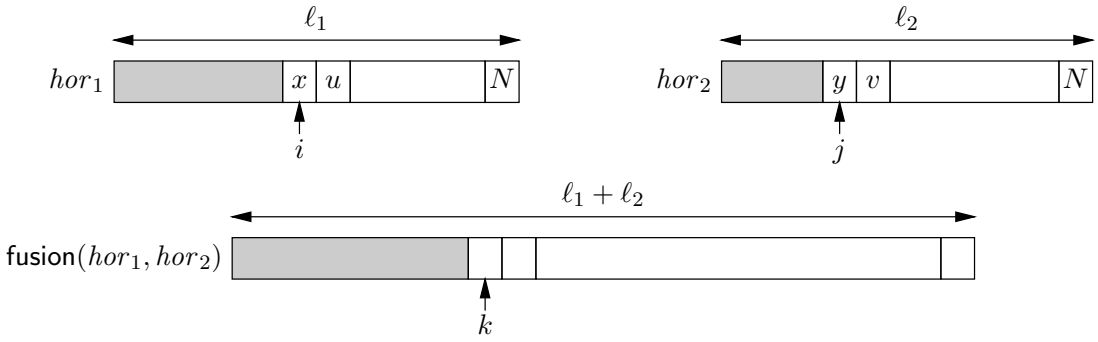
$$hor[2k] < hor[2k+2] \ \ (0 \leq k < (\ell-1)/2) \ \text{ et } \ hor[2k+1] \neq hor[2k+3] \ \ (0 \leq k < (\ell-3)/2)\ .$$

Question 4. Écrire une fonction canonique(hor) qui retourne un tableau représentant hor en forme canonique, en temps linéaire par rapport à la longueur ℓ de hor.

On veut à présent fusionner deux lignes d'horizon comme indiqué sur la figure suivante :



Les deux lignes d'horizon sont représentées par les tableaux hor_1 et hor_2 de longueurs ℓ_1 et ℓ_2. On peut effectuer cette fusion très facilement en examinant les deux tableaux de la gauche vers la droite, en maintenant la hauteur u' du dernier bâtiment rencontré dans hor_1 et la hauteur v' du dernier bâtiment rencontré dans hor_2, comme indiqué sur la figure suivante :



Question 5. Écrire la fonction fusion (hor_1, hor_2) qui retourne, en temps linéaire par rapport à $\ell_1 + \ell_2$, une ligne d'horizon fusionnant les lignes d'horizon hor_1 et hor_2 de longueur ℓ_1 et ℓ_2.

* *
*

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2006

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR **FILIÈRE PC**

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Disque dur à deux têtes

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. On précisera en tête de copie le langage de programmation utilisé.

Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute :

en temps $O(n^\alpha)$, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn^\alpha$.

Ce problème étudie des stratégies de déplacement des têtes d'un disque dur afin de minimiser le temps moyen d'attente entre deux requêtes au disque dur (de lecture ou d'écriture). Dans ce problème, le disque dur est représenté par une demi-droite $[0, +\infty)$ et possède deux têtes de lecture/écriture. Chacune des têtes peut aller indifféremment à n'importe quelle position sur le disque pour y lire ou écrire une donnée. Les deux têtes peuvent être au même endroit ou encore se croiser. On ne s'intéresse qu'aux temps de déplacement des têtes et non aux temps de lecture/écriture. Les deux têtes ont la même vitesse de déplacement. Le temps de déplacement d'une tête est supposé égal à la distance qu'elle parcourt.

Une requête r est un entier positif ou nul représentant l'emplacement du disque auquel l'une des deux têtes doit se rendre. Initialement les deux têtes sont chacune à la position 0.

Le disque dur est muni d'une mémoire (appelée cache) qui permet d'enregistrer n requêtes ($n > 0$) avant de les traiter. À chaque bloc de n requêtes présentes dans le cache, le contrôleur du disque dur doit alors satisfaire ce bloc de requêtes, dans leur *ordre* d'arrivée, en minimisant le déplacement *total* des deux têtes. L'ordre importe puisqu'une opération d'écriture peut précéder une autre opération de lecture ou d'écriture. Il faut donc déterminer pour chacune des n requêtes le numéro de la tête à déplacer de manière à minimiser la somme totale des temps de tous les déplacements.

Notes de programmation : On supposera que le langage utilisé permet de définir des fonctions qui retournent des tableaux. On pourra aussi supposer que le nombre de requêtes n

est une constante du programme. Enfin, une autre constante du programme `Infini` $= -1$ sera utilisée pour coder l'infini, noté ∞ dans la Partie II.B.

Partie I

A. Coût d'une séquence de déplacements

Un bloc de n requêtes est représenté par une suite de n entiers positifs ou nuls $\langle r_1, r_2, \dots r_n \rangle$ rangés dans un tableau r de taille n. Une *séquence de déplacements* $\langle d_1, d_2, \dots d_n \rangle$ est une suite de n entiers, 1 ou 2, rangés dans un tableau d indiquant à l'étape i qui de la première tête ($d_i = 1$) ou de la deuxième tête ($d_i = 2$) doit se déplacer à la position r_i ($1 \leq i \leq n$). Le *coût* d'une séquence de déplacements est la somme totale des distances parcourues par chacune des têtes.

Ainsi pour le bloc de requêtes $\langle 5, 2, 4 \rangle$, le coût de la séquence de déplacements $\langle 1, 1, 2 \rangle$ est $5 + 3 + 4 = 12$, alors que le coût de $\langle 1, 2, 1 \rangle$ vaut $5 + 2 + 1 = 8$.

Question 1 Écrire une fonction coutDe(r, d) qui calcule le coût d'une séquence de déplacements d pour le bloc de requêtes r.

Le coût optimal d'une suite de requêtes r est le plus petit coût des séquences de déplacements satisfaisant le bloc de requêtes r.

Question 2 Montrer qu'il existe toujours une séquence de déplacements de coût optimal qui commence par 1, c'est-à-dire commençant par déplacer la première tête.

Question 3 Combien de séquences de déplacements satisfont un bloc de requêtes r donné ?

B. Coût optimal pour deux requêtes

Dans cette partie, le cache est de taille 2 ($n = 2$). Il n'y a donc que deux requêtes r_1 et r_2. Par convention, la première tête sera toujours celle qui bouge sur la première requête.

Question 4 Donner une séquence de déplacements de coût minimal pour chacun des deux blocs de requêtes $\langle 10, 3 \rangle$ et $\langle 3, 10 \rangle$.

Question 5 Écrire une fonction coutOpt2(r_1, r_2) qui retourne un tableau d, de longueur 2, donnant une séquence de déplacements de coût optimal.

Partie II

A. Coût optimal pour trois requêtes

Dans cette partie, le cache est de taille 3 ($n = 3$). Il y a donc trois requêtes r_1, r_2 et r_3. Par convention, la première tête sera toujours celle qui bouge sur la première requête.

Question 6 On suppose que la fonction de la question 5 a été étendue au cas de trois requêtes en appliquant la même règle de décision à la troisième requête qu'à la deuxième requête. L'appliquer en justifiant sur l'exemple $\langle 20, 9, 1 \rangle$.

Question 7 Énumérer toutes les stratégies possibles sur l'exemple de la question précédente. En déduire que l'approche de la question 6 ne fournit pas la solution de coût minimal.

Question 8 Écrire une fonction coutOpt3(r_1, r_2, r_3) qui retourne un tableau d donnant une séquence de déplacements de coût optimal.

B. Coût optimal pour n requêtes

Dans cette partie, on calcule le coût minimal sans pour autant trouver une séquence de déplacements donnant ce coût. Par commodité, chacune des deux têtes peut effectuer indifféremment le premier déplacement.

On pose $r_0 = 0$ pour coder la position initiale des têtes. À un instant donné, la configuration des têtes du disque dur est représentée par une paire (i, j) codant le numéro des deux dernières requêtes respectivement satisfaites par chacune des deux têtes : la première tête a satisfait en dernier la $i^{ième}$ requête et la deuxième tête la $j^{ième}$ requête. Par convention, la configuration initiale est $(0, 0)$.

À chaque requête r_k, on associe la matrice $(n+1) \times (n+1)$ représentée par le tableau d'entiers à deux dimensions cout_k. L'élément $\mathsf{cout}_k[i][j]$ est égal au coût optimal pour atteindre la configuration (i, j), après avoir satisfait la $k^{ième}$ requête. On pose $\mathsf{cout}_k[i][j] = \infty$ si cette configuration n'est pas accessible.

Question 9 Expliquer comment calculer le coût optimal d'une suite de requêtes $\langle r_1, r_2, \dots, r_n \rangle$ à l'aide du tableau correspondant cout_n.

Question 10 Montrer que les matrices $(\mathsf{cout}_k)_{0 \leq k \leq n}$ satisfont :
$\mathsf{cout}_0[0][0] = 0$ et $\mathsf{cout}_0[i][j] = \infty$ pour tout $i \neq 0$ ou $j \neq 0$;
$\mathsf{cout}_k[i][k]$ est le minimum de $|r_k - r_j| + \mathsf{cout}_{k-1}[i][j]$ pour $0 \leq j \leq n$;
$\mathsf{cout}_k[k][j] = \mathsf{cout}_k[j][k]$;
$\mathsf{cout}_k[i][j] = \infty$ si $i \neq k$ et $j \neq k$.

Question 11 Écrire une procédure mettreAJour(cout, r, k) qui met à jour le tableau cout en fonction de la nouvelle requête r_k, de sorte que si cout contenait les valeurs du tableau cout_{k-1}, alors, après la mise à jour, cout contient les valeurs du tableau cout_k.

Question 12 En déduire une fonction coutOpt(r) permettant de trouver le coût minimal du bloc de n requêtes r. Donner le temps d'exécution de coutOpt(r) par rapport à n.

La matrice cout est très creuse. Après avoir satisfait la $k^{ième}$ requête, seule la $k^{ième}$ ligne et la $k^{ième}$ colonne peuvent contenir des valeurs différentes de ∞. De plus, comme la matrice cout est symétrique, seule la $k^{ième}$ ligne est à retenir.

Question 13 Écrire une nouvelle fonction coutOpt(r) qui calcule le coût minimal du bloc de n requêtes r en n'utilisant qu'un tableau cout à une dimension de taille $n+1$. Évaluer son nouveau temps d'exécution.
Indication : On remarquera que pour parvenir à la configuration (i, k), avec $i < k-1$, nécessairement on doit venir de la configuration $(i, k-1)$, en revanche pour la configuration $(k-1, k)$ on peut provenir de n'importe quelle configuration $(k-1, j)$.

* *
*

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS 2004

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR FILIÈRE PC

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Compression ternaire

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. On supposera que le langage de programmation utilisé possède deux opérations x `div` y et x `mod` y donnant le quotient et le reste de la division euclidienne de x par y.

Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute :

- en temps linéraire en n, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn$;
- en temps quadratique en n, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leq Kn^2$.

Nombres ternaires

En base 3, les entiers 0, 1, 2, 3, 4, 5, 6, 7, 8 sont représentés par 00, 01, 02, 10, 11, 12, 20, 21, 22. Le chiffre de poids fort de `bc` est `b` ; le chiffre de poids faible est `c`.

Question 1. Écrire la fonction `entier(b, c)` retournant l'entier compris entre 0 et 8 qui s'écrit `bc` en base 3.

Question 2. Soit x un entier vérifiant $0 \leq x \leq 8$. Écrire une fonction poidsFort(x) retournant le chiffre de poids fort de x en base 3. Écrire la fonction poidsFaible(x) retournant le chiffre de poids faible de x.

Textes ternaires

Dans ce problème, les textes sont représentés en représentation ternaire. Un savant russe nous a convaincus de la pertinence de ce choix plus compact que la représentation binaire. Un texte est rangé dans un tableau t de N caractères vérifiant $t[i] \in \{0,1,2\}$ pour tout i vérifiant $0 \leq i < N-1$; par ailleurs $t[N-1] = X > 2$ (le dernier caractère n'est pas ternaire). On suppose $N \geq 1$.

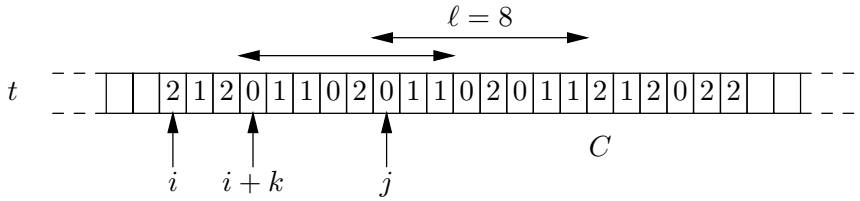
Quelques définitions sont nécessaires : la chaîne de caractères de longueur ℓ démarrant en i est la suite $\langle t[i], t[i+1], \ldots t[i+\ell-1]\rangle$. On dira que deux chaînes $\langle t[i], t[i+1], \ldots t[i+\ell-1]\rangle$ et $\langle t[j], t[j+1], \ldots t[j+\ell'-1]\rangle$ sont égales si $\ell = \ell'$ et $t[i+k] = t[j+k]$ pour $0 \leq k < \ell$.

Question 3. Écrire une fonction longueurMotif(t, i, j, m) qui retourne, en temps linéaire par rapport à N, la plus grande longueur ℓ d'une chaîne démarrant en i égale à une chaîne démarrant en j. En outre, cette longueur doit vérifier $\ell \leq m$.

Question 4. On suppose $i < j$. Écrire une fonction longueurMotifMax(t, i, j, m) qui retourne, en temps quadratique par rapport à N, la plus grande longueur ℓ d'une chaîne démarrant en $i+k$ égale à une chaîne démarrant en j pour $0 \leq k < m$. En outre, on exige $i+k < j$ et $\ell \leq m$.

On suppose qu'il existe trois variables globales entières A, L, C.

Question 5. Modifier la fonction précédente pour obtenir la fonction motifMax(t, i, j, m) qui rend, en temps quadratique, dans L la plus grande longueur ℓ d'une chaîne démarrant en $i+k$ égale à une chaîne démarrant en j pour $0 \leq k < m$; qui rend dans A la valeur de k pour lequel $i+k$ est l'indice de départ de cette chaîne de longueur maximale ; qui rend enfin dans C le caractère suivant cette chaîne à partir de j dans t. À nouveau, cette longueur doit vérifier $\ell \leq m$. Et on a $i+k < j$ (cf. l'exemple dans la figure suivante).



$A = k = 3;\ L = \ell = 8;\ C = 2$

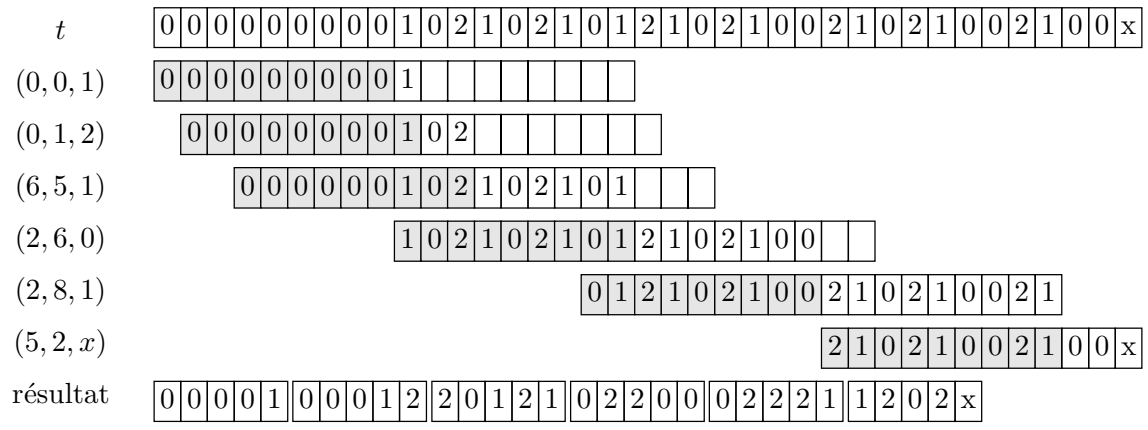
Compression

La méthode de compression de Ziv et Lempel, adoptée dans les commandes `zip` ou `gzip`, consiste à repérer les motifs maximaux déjà rencontrés dans un texte et à indiquer pour chacun d'eux le triplet (A, L, C) calculé dans la question précédente entre toute paire d'indices i et j. Pour mesurer le facteur de compression, nous utilisons le même codage pour ces triplets que pour les caractères du texte, c'est-à-dire le système ternaire dans ce problème.

Question 6. Écrire une fonction imprimerTriplet(A, L, C) qui imprime les arguments A, L, C sous forme de cinq chiffres consécutifs, les deux caractères ternaires de A, puis les deux caractères ternaires de L, puis le chiffre C, en imposant $0 \leq A < 9, \quad 0 \leq L < 9 \quad$ et $\quad 0 \leq C \leq 9$.

On suppose à présent que t contient un long texte ternaire commençant par 9 caractères 0 ; en outre, t finit par un caractère `x` spécial ($x > 2$). On déplace sur ce texte une fenêtre de longueur 18. Au début, cette fenêtre est alignée à gauche sur le début du tableau, et on pose $j = 9$. En régime de croisière, la dixième case de la fenêtre correspond à l'entrée j du tableau t. On recherche, dans la partie gauche de la fenêtre, la chaîne de longueur ℓ maximale vérifiant $\ell < 9$ et égale à une chaîne de caractères démarrant en j. On imprime, grâce à la fonction imprimerTriplet, le triplet (A, L, C) donné par motifMax. Puis, on recentre la fenêtre sur le caractère suivant le caractère d'arrêt C. Ce processus continue jusqu'au bout du tableau t comme indiqué par la figure.

Ainsi pour le texte suivant, on obtient les décompositions de chacune des lignes, soit au total le facteur de compression 30/37 qui serait nettement meilleur dans une base supérieure à 3 et si la taille de la fenêtre était plus grande que 18 (Il y a en effet 30 caractères dans le résultat et 37 dans le texte d'entrée).



Question 7. Écrire une fonction compresser(t) qui imprime sur le terminal de sortie le texte compressé par la méthode précédente.

Pour la décompression, on produit d'abord 9 caractères 0. On considère ensuite tous les triplets (A, L, C) représentés par 5 caractères ternaires consécutifs et on recrée la chaîne originale jusqu'au dernier triplet dont la composante C n'est pas comprise entre 0 et 2.

Question 8. Écrire une fonction décompresser(tc) qui prend un texte ternaire tc correspondant à du texte compressé et imprime sur le terminal de sortie le texte décompressé correspondant.

* *
*

ANNEXE

D

Intégration numérique

Il est très rare de pouvoir déterminer la dérivée ou une primitive formelle d'une fonction donnée. Il faut recourir à des méthodes numériques dont l'utilisation entraîne de nouveaux problèmes : il faut pouvoir contrôler l'approximation de manière théorique et ne pas oublier que cela sera fait la plupart du temps par une machine qui elle aussi apporte son lot d'imprécision.

Werner Romberg (1909 - 2003) né à Berlin, s'installe en Norvège en 1938. Dix-sept ans plus tard, il publie un article se basant sur l'extrapolation de Lewis Richardson (1881 - 1953) et qui accélère grandement l'efficacité de la méthode dite des trapèzes utilisée depuis longtemps pour calculer des approximations d'intégrales. C'est cette méthode que nous étudierons aujourd'hui en la comparant aux méthodes de Newton-Cotes, Gauss-Legendre, Kuncir après une introduction sur les méthodes de dérivation numérique.

Méthode de quadrature simplifiée déterministe

On étudiera des formules d'approximation reposant sur des formules du type :

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=1}^{n} w_{i,n} f(x_i) + E_n(f)$$

où les w_i sont les *poids*, les x_i sont les pivots et $E(f)$ l'erreur commise.
Une méthode de quadrature est associée à un ordre :

Définition D - 1

Une méthode de quadrature est d'ordre m si elle est exacte pour tous les polynômes de $\mathbb{R}_m[X]$.

Nous admettrons le théorème suivant qui donne une CNS de convergence d'une méthode de quadrature :

Théorème D - 1

Pour toute fonction continue sur $]a,b[$, $\lim\limits_{n\to+\infty} E_n(f) = 0$ si, et seulement si,

— $(\exists M \in \mathbb{R}_+^\star)(\forall n \in \mathbb{N})(\sum\limits_{i=0}^{n} |w_{i,n}| \leqslant M)$

— $(\forall k \in \mathbb{N})(\lim\limits_{n\to+\infty} E_n(x \mapsto x^k) = 0\ )$

On peut se contenter, pour la condition suffisante, d'avoir chaque $w_{i,n}$ positif et f continue par morceaux.
Ce qui nous intéressera au plus haut point est le théorème suivant qui donne une estimation de l'erreur commise :

Théorème D - 2

On suppose que la méthode est d'ordre m et $f \in \mathcal{C}^{(m+1)}([a,b],\mathbb{R})$. Alors

$$E(f) = \frac{1}{m!}\int_a^b K_m(x) f^{(m+1)}\,\mathrm{d}x$$

avec $K_n(x) = E\left(x \mapsto \max\left((x-t)^n, 0\right)\right)$. La famille $(K_n)_{n\in\mathbb{N}}$ s'appelle la famille des noyaux de Peano associée à la méthode considérée.

On utilise la formule de Taylor intégrale puis le fait que la méthode est d'ordre m.
On en déduit le corollaire suivant :

Théorème D - 3

Si K_m est de signe constant sur $[a,b]$:

$$(\exists \xi \in]a,b[)(E(f) = \frac{1}{(m+1)!} f^{(m+1)}(\xi) E(x \mapsto x^{m+1}))$$

On utilise la deuxième formule de la moyenne.

Méthodes de Newton-Cotes composées

Pour donner une valeur approchée de l'intégrale I, une première idée est de remplacer la fonction f par un polynôme P qui interpole f en plusieurs points. Cependant, dès qu'on augmente le nombre de pivots, on risque fort de se retrouver confronté au phénomène de Runge.
On modifie alors l'idée de départ ainsi : on commence par subdiviser régulièrement l'intervalle d'intégration en n sous-intervalles $[x_j, x_{j+1}]$ où

$$x_0 = a, \quad x_n = b, \quad \text{et} \quad \forall j \in [\![0,n]\!], \quad x_j = a + jh, \quad \text{avec } h = \frac{b-a}{n}$$

Ensuite, on remplace f par un polynôme P_j qui interpole f sur chacun des petits segments $[x_j, x_{j+1}]$.

— Si P_j interpole f au point x_j, alors le graphe de P_j est une droite horizontale : c'est la méthode des rectangles.

— Si P_j interpole f aux points x_j et x_{j+1}, alors le graphe de P_j est une droite affine : c'est la méthode des trapèzes.

— Si P_j interpole f aux points x_j, $\frac{x_j+x_{j+1}}{2}$ et x_{j+1}, alors le graphe de P_j est une parabole : c'est la méthode de SIMPSON 1/3.

Enfin, on somme les intégrales de chaque polynôme P_j sur $[x_j, x_{j+1}]$, et on obtient une valeur approchée de I :

$$I = \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} f(x)\,\mathrm{d}x = \sum_{j=0}^{n-1} \left(\int_{x_j}^{x_{j+1}} P_j(x)\,\mathrm{d}x + E_j \right) \approx \sum_{j=0}^{n-1} \int_{x_j}^{x_{j+1}} P_j(x)\,\mathrm{d}x$$

On se place sur l'intervalle $[x_j, x_{j+1}]$ et on note $\xi_j = \frac{x_j+x_{j+1}}{2}$; alors d'après les formules des différences divisées (cf. TP précédent...), le polynôme d'interpolation de f aux points (x_j, ξ_j, x_{j+1}) est donné par

$$P_j(x) = f[x_j] + f[x_j, \xi_j](x - x_j) + f[x_j, \xi_j, x_{j+1}](x - x_j)(x - \xi_j)$$

On en déduit, après calculs,

$$\int_{x_j}^{x_{j+1}} P_j(x)\,\mathrm{d}x = \frac{h}{6}\left(f(x_j) + 4f(\xi_j) + f(x_{j+1})\right)$$

Par sommation on obtient

$$\begin{aligned} I &\approx \frac{b-a}{6n}\left(f(x_0) + 4f\left(\tfrac{x_0+x_1}{2}\right) + 2f(x_1) + 4f\left(\tfrac{x_1+x_2}{2}\right) + \cdots + 2f(x_{n-1}) + 4f\left(\tfrac{x_{n-1}+x_n}{2}\right) + f(x_n)\right) \\ &\approx \frac{b-a}{6n}\left(f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + 4\sum_{j=0}^{n-1} f\left(\tfrac{x_j+x_{j+1}}{2}\right) + f(x_n)\right) \\ &\approx \frac{b-a}{6n}\sum_{k=0}^{2n} w_k f(a_k) \quad \text{avec} \quad a_k = a + k\cdot\frac{b-a}{2n} \text{ et } w_k = \begin{cases} 1 \text{ si } k = 0 \text{ ou } 2n \\ 4 \text{ si } 0 < k < 2n \text{ et } k \text{ impair} \\ 2 \text{ si } 0 < k < 2n \text{ et } k \text{ pair} \end{cases} \end{aligned}$$

On peut montrer que l'erreur commise est majorée par

$$|E| \leqslant \frac{1}{2880}\cdot h^4 \cdot \left\|f^{(4)}\right|_\infty \cdot (b-a) \qquad (**)$$

On dit alors que la méthode est d'ordre 4 ; noter que la méthode est exacte pour tout polynôme de degré au plus 3.

Recherche

Programmez les trois méthodes mentionnées : **`Rectangles`**, **`Trapezes`** et **`Simpson`**.
On introduira la fonction $x \mapsto \frac{4}{1+x^2}$ et on comparera son intégrale sur [0,1] avec π en utilisant les différentes méthodes.

En pratique, il est difficile d'avoir un majorant de la dérivée quatrième d'une fonction. On préfère donc en pratique appliquer la méthode avec deux pas différents (h et $2h$ pour minimiser les évaluations de f) et on utilise la différence des deux approximations numériques comme estimation de l'erreur du moins bon résultat.
En prenant 5 points d'interpolation, on construit la méthode dite de BOOLE-VILLARCEAU et avec 7 celle de WEDDLE-HARDY. On ne va pas plus loin car sinon on obtient des poids négatifs qui amplifient les erreurs d'arrondi.

La méthode de Romberg

La méthode de ROMBERG utilise le procédé d'extrapolation de RICHARDSON pour accélérer la convergence de la méthode des trapèzes composée.
Soit f une fonction de classe $\mathcal{C}^\infty$ sur le segment $[a,b]$. On décompose le segment $[a,b]$ en n sous-intervalles égaux. Alors l'approximation de l'intégrale I de f sur ce segment par la méthode des trapèzes vaut

$$T(h) = h\left(\frac{1}{2}f(x_0) + \sum_{j=1}^{n-1} f(x_j) + \frac{1}{2}f(x_n)\right) \quad \text{où} \quad \forall j \in [\![0,n]\!], \quad x_j = a + jh, \quad \text{et} \quad h = \frac{b-a}{n}$$

On peut alors montrer à l'aide de la formule d'EULER-MACLAURIN que l'erreur commise dans la méthode des trapèzes possède un développement limité à tout ordre de la forme

$$E(h) = a_1h^2 + a_2h^4 + \cdots + a_{k-1}h^{2k-2} + O(h^{2k}) \quad \text{où} \quad E(h) = T(h) - \int_a^b f(t)\,\mathrm{d}t$$

De plus, les a_i ne dépendent ni de n, ni de h.

L'idée est alors de faire une combinaison linéaire de $T(h)$ et $T(h/2)$ pour annuler le premier terme du développement de E :

$$\begin{aligned} T(h) &= \int_a^b f(t)\,\mathrm{d}t + a_1 h^2 + a_2 h^4 + \cdots + a_{k-1}h^{2k-2} + O(h^{2k}) \\ T\left(\tfrac{h}{2}\right) &= \int_a^b f(t)\,\mathrm{d}t + a_1 \tfrac{h^2}{2^2} + a_2 \tfrac{h^4}{2^4} + \cdots + a_{k-1}\tfrac{h^{2k-2}}{2^{2k-2}} + O(h^{2k}) \\ \hline \tfrac{1}{3}\left(4T\left(\tfrac{h}{2}\right) - T(h)\right) &= \int_a^b f(t)\,\mathrm{d}t + \underbrace{b_2 h^4 + \cdots + b_{k-1}h^{2k-2} + O(h^{2k})}_{O(h^4)} \end{aligned}$$

On remarque qu'ainsi, on a gagné un ordre dans le développement de l'erreur. C'est un bon exercice de vérifier que la nouvelle expression obtenue correspond à la méthode de SIMPSON.
Le procédé précédent se généralise aisément : on utilise des dichotomies successives avec des pas de la forme $h = \frac{b-a}{2^m}$ et on remplit un tableau triangulaire comprenant les nombres :

$$A_{m,0} = T\left(\frac{b-a}{2^m}\right) \quad \text{et} \quad \forall n \in [\![1, m]\!],\ A_{m,n} = \frac{4^n A_{m,n-1} - A_{m-1,n-1}}{4^n - 1}$$

étape	$t=0$	$t=1$	$t=2$	$\cdots$	$t=m$
tableau[0]	$A_{0,0}$				
tableau[1]	$A_{1,0}$	$A_{1,1}$			
tableau[2]	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
tableau[n]	$A_{m,0}$	$A_{m,1}$	$A_{m,2}$	$\cdots$	$A_{m,m}$

Le dernier élément de la diagonale principale $A_{m,m}$ est alors la meilleure approximation de l'intégrale I que l'on puisse obtenir en évaluant f en 2^m points.
En pratique, on commence par remplir la première colonne en se servant de la relation de récurrence :

$$A_{k,0} = \frac{1}{2}A_{k-1,0} + A'_{k,0} \quad \text{où} \quad A'_{k,0} = \frac{b-a}{2^k}\sum_{j=1}^{2^{k-1}} f\left(a + (2j+1)\frac{b-a}{2^k}\right)$$

En effet, en notant $h = \frac{b-a}{2^k}$, on a

$$\begin{aligned} A_{k,0} &= h\left(\frac{1}{2}f(a) + \sum_{j=1}^{2^k-1} f(a+jh) + \frac{1}{2}f(b)\right) \\ &= h\left(\frac{1}{2}f(a) + \sum_{p=1}^{2^{k-1}-1} f(a+2ph) + \sum_{p=0}^{2^{k-1}-1} f(a+(2p+1)h) + \frac{1}{2}f(b)\right) \\ &= \frac{1}{2}A_{k-1,0} + \underbrace{h\sum_{j=0}^{2^{k-1}-1} f(a+(2j+1)h)}_{A'_{k,0}} \end{aligned}$$

Ceci permet de calculer une fois et une fois seulement les valeurs de f nécessaires.
Ensuite, on remplit les autres colonnes en utilisant la formule

$$\forall n \in [\![1, m]\!],\ A_{m,n} = \frac{4^n A_{m,n-1} - A_{m-1,n-1}}{4^n - 1}$$

On montre alors par récurrence que :

$$A_{m,k} = \int_a^b f(x)\,\mathrm{d}x + O\left(\frac{1}{2^{m(2k+2)}}\right)$$

Recherche

Programmez Python pour obtenir $A_{m,m}$.
On estime l'erreur comme d'habitude à $|A_{m,m} - A_{m-1,m-1}|$.

ANNEXE F

Discrétisation d'équations différentielles

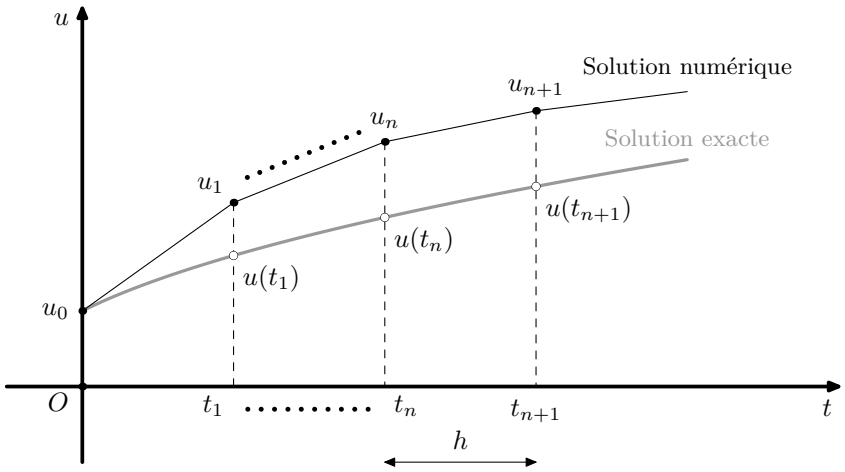
La résolution formelle d'équations différentielles s'avère très compliquée et limitée : la plupart des problèmes ne peuvent être qu'approchés.
Le très prolifique mathématicien suisse Leonard Euler mit au point au XVIII^{e} siècle une méthode de discrétisation des équations différentielles que généralisèrent et améliorèrent les Allemands Carl Runge (1856-1927) dont vous pouvez admirer le portrait, et Martin Kutta (1867-1944). Nous allons donc étudier ces méthodes de Runge-Kutta très utiles pour l'ingénieur.
Il est à noté que Runge mit également en évidence des fonctions assez régulières faisant diverger la méthode d'interpolation polynomiale de Lagrange et donna son nom à ce phénomène.

Principe

On considère une équation différentielle ordinaire (EDO pour les intimes) $u'(t) = f(t, u(t))$. Lorsqu'on ne connait pas de solution exacte à cette EDO, on essaye d'en avoir une bonne approximation par des méthodes numériques.

Nous allons par exemple « observer » u à intervalle de temps régulier : notons h cette période d'échantillonnage.

Nous allons essayer d'obtenir une suite d'approximations u_n de $u(t_n)$ et d'évaluer la pertinence de cette approximation, le principe étant résumé sur la figure suivante :



Nous considérerons des fonctions f K-lipschitziennes en chacune des variables et des fonctions u de classe $\mathcal{C}^1$ d'un intervalle I dans $\mathbb{R}$ avec une condition initiale $u(t_0) = y_0$.

Pour une étude mathématique complète des phénomènes de convergence, on pourra étudier l'article http://www.math.ens.fr/~debarre/Exponentielle-Log.pdf, lire le chapitre 5 de Modélisation à l'oral de l'agrégation de Laurent Dumas et les chapitres 4 et 5 de l'ouvrage Analyse numérique des équations différentielles par Michel Crouzeix et Alain Mignot et surtout « Solving Ordinary Diffrential Equations » de Hairer, Norsett et Wanner chez Springer.

On notera I_n l'intervalle $[t_n, t_{n+1}]$ avec $t_{n+1} = t_n + h$, la suite (t_n) constituant une subdivision régulière de l'intervalle I.

On cherche donc à construire une suite finie (u_n) telle que $u_0 = y_0$ et $u_{n+1} = u_n + h\varphi(t_n, u_n)$ (format explicite à un pas) ou $u_{n+1} = u_n + h\varphi(t_{n+1}, u_{n+1})$ (format implicite à un pas).

Méthode d'Euler explicite

Intégrons l'équation différentielle $u'(t) = f(t, u(t))$ sur l'intervalle I_n :

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(s, u(s))\,\mathrm{d}s$$

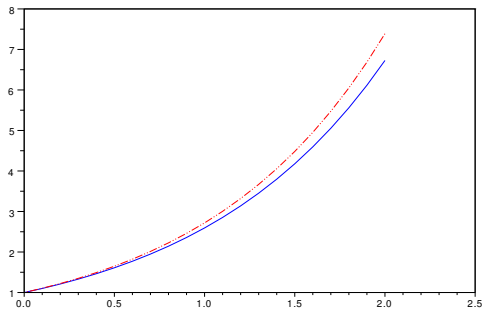
L'utilisation de méthodes de quadratures vues au TP précédent vont nous permettre de calculer u_{n+1} connaissant u_n.

La méthode des rectangles à gauche donne le schéma d'Euler explicite.

Déterminer une fonction Python qui renvoie la liste des approximations des abscisses et des ordonnées de la série de points de la solution d'une équation différentielle par le schéma d'Euler explicite.

On pourra créer par exemple `euler_imp(f,a,b,N,yo)`.

On pourra comparer avec des cas connus.



Méthode d'Euler modifiée

Nous avons vu au TP précédent que la méthode d'intégration du point milieu nous faisait gagner un ordre de précision.
Appliquez alors cette méthode à notre problème.
On pourra alors remarquer qu'avec $N = 4096$ la méthode d'EULER a une précision de l'ordre de :

Python
```
0.0036059
```

et avec la méthode d'EULER modifiée :

Python
```
0.0000006
```

Méthode RK4

On utilise cette fois la méthode de SIMPSON. Notons :

$$S = \frac{h}{6}\left(f(t_n, u(t_n)) + 4f(t_n + h/2, u(t_n + h/2)) + f(t_n + h, u(t_n + h))\right)$$

Alors faisons les approximations suivantes :

$$u(t_n) \to u_n = U_0$$

puis :

$$u(t_n + \frac{h}{2}) \to u_n + \frac{h}{2} f\left(t_n, u_n\right) = U_1$$

en utilisant le rectangle gauche, ou bien :

$$u(t_n + \frac{h}{2}) \to u_n + \frac{h}{2} f\left(t_n + \frac{h}{2}, u(t_n + \frac{h}{2})\right) \to u_n + \frac{h}{2} f\left(t_n + \frac{h}{2}, U_1\right) = U_2$$

en utilisant le rectangle droit.
On remplace alors $f(t_n + h/2, u(t_n + h/2))$ par la moyenne de U_1 et U_2
Enfin :

$$u(t_n + h) \to u_n + \frac{h}{2}\left(t_n + \frac{h}{2}, u(t_n + \frac{h}{2})\right) \to u_n + \frac{h}{2}\left(t_n + h/2, U_2\right) = U_3$$

Alors :

$$6S \to hf\left(t_n, u_n\right) + 4h\frac{f(t_n + h/2, U_1) + f(t_n + h/2, U_2)}{2} + hf(t_n + h, U_3)$$

On note conventionnellement :
$k_1 = hf\left(t_n, u_n\right)$
$k_2 = hf\left(t_n + h/2, u_n + k_1/2\right)$
$k_3 = hf\left(t_n + h/2, u_n + k_2/2\right)$
$k_4 = hf\left(t_n + h, u_n + k_3\right)$
Alors

$$u_{n+1} = u_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

Programmez cette nouvelle méthode. Quelle est alors la précision pour la fonction habituelle avec $N = 4096$?

La boîte de Pandore des approximations

Encore une fois, dès que vous tentez d'approcher une solution, de nombreux facteurs peuvent en fait vous en éloigner : une étude mathématique sophistiquée doit alors être envisagée (vous verrez ça en master...)
Voici quelques exemples apparemment anodins tirés de la littérature.

Problème mal posé mathématiquement

Que pensez-vous du problème :

$$\begin{cases} u'(t) = 2\sqrt{|u(t)|} \\ u(0) = 0 \end{cases}$$

Problème mal posé numériquement

Considérons le problème :

$$\begin{cases} u'(t) = 5u(t) - 5, \quad t \in [0, 50] \\ u(0) = 1 \end{cases}$$

et imaginons une petite perturbation de la valeur initiale : $u(0) = 1 + \varepsilon$. Estimez $\widetilde{u}(50) - u(50)$ pour $\varepsilon = 10^{-17}$ (de l'ordre de l'epsilon de Python).

Problème mal conditionné

Cette fois, le problème est le suivant :

$$\begin{cases} u'(t) = -150u(t) - 30 \\ u(0) = 1/5 \end{cases}$$

Vérifiez qu'il est bien posé mathématiquement et numériquement.
Approchons-le maintenant à l'aide de la méthode d'Euler explicite.
Vérifiez qu'on obtient

$$u_n = \frac{1}{5} + \left(1 - \frac{150}{N}\right)^n \left(u_0 - \frac{1}{5}\right)$$

Qu'obtenez-vous pour $N = 50$ et $u_0 = \frac{1}{5} + \varepsilon$?
Observez maintenant avec nos fonctions Python avec les différentes méthodes vues précédemment.

Problèmes raides

Il existe des problèmes qui résistent à toutes les méthodes usuelles. On trouve par exemple ces phénomènes en chimie où deux réactions ont des échelles de temps très différentes.
Considérez par exemple :

$$\begin{cases} u'(t) = -1000(u(t) - \exp(-t)) - \exp(-t) \\ u(0) = 0 \end{cases}$$

et testez vos fonctions sur $[0, 1]$.
Cependant, vous pouvez utiliser la fonction **ode** de Python qui utilise au mieux les méthodes précédentes en adaptant de plus les pas ce qui permet parfois d'obtenir de bons résultats.
La doc nous dit en effet :

> *Le solveur lsoda du package ODEPACK est utilisé par défaut. Il choisit automatiquement entre un schéma prédicteur-correcteur d'Adams et un schéma adapté au systèmes raides (stiff) de type « Backward Differentiation Formula » (BDF).*
>
> *Initialement le schéma adapté aux systèmes non raides est choisi puis la méthode adaptée est ensuite choisie dynamiquement.*

Pour le problème mal conditionné du paragraphe précédent :

Python

```
uo = 1/5+1E-15;
to = 0;
u = ode(uo,to,t,f);
```

Systèmes différentiels et ordre 2

Systèmes

Comme Python travaille sur des vecteurs, on peut considérer des fonctions dans $\mathbb{R}^2$ de manière naturelle. Supposons que nous voulions résoudre le système :

$$\begin{cases} u_1'(t) = 3u_1(t) - 4u_2(t) \\ u_2'(t) = u_1(t) + 3u_2(t) \\ u_1(0) = 2 \\ u_2(0) = 0 \end{cases}$$

On utilise la même syntaxe que précédemment...mais avec une fonction $u : \mathbb{R} \to \mathbb{R}^2$.
Comparez avec la solution attendue.

Ordre 2

On considère à présent l'équation bien connue du pendule pesant amorti : $y'' + y' + \sin(y) = 0$ avec $y(0) = 0$ et $y'(0) = 1$.
Posez $u = {}^t(y, y')$ et essayez de revenir à un système différentiel que vous résoudrez sur $[0, 10]$.
Vous tracerez le portrait de phase.

CCP MP : Sujet 0 de SI 2015

Après 25 questions, voici l'éternelle, l'inévitable question sur la méthode d'Euler... Apparemment, pour un mécanicien, le rôle de l'informatique se résume à donner des solutions approchées d'une équation différentielle...
Nous sommes donc à la page 14 du sujet :-) On introduit l'inévitable équa diff.
[On obtient] l'équation différentielle suivante, liant la vitesse de rotation du moteur $\omega_m(t)$ à la vitesse de consigne $\omega_{\text{const}}(t)$:

$$\frac{1}{\omega_0^2}\frac{\mathrm{d}^2\omega_m(t)}{\mathrm{d}t^2} + \frac{2m}{\omega_0}\frac{\mathrm{d}\omega_m(t)}{\mathrm{d}t} + \omega_m(t) = K\omega_{\text{const}}(t)$$

Dans la suite, on s'intéressera uniquement à la réponse indicielle unitaire du système dans les conditions de Heaviside, ce qui revient à prendre $\omega_{\text{const}}(t) = u(t)$ avec $\omega_m(0) = 0$ et $\omega_m'(0) = 0$.

En posant le vecteur $\boldsymbol{Y}$ tel que $\boldsymbol{Y}(t) = \begin{pmatrix} \omega_m(t) \\ \frac{\mathrm{d}\omega_m(t)}{\mathrm{d}t} \end{pmatrix}$, l'équation différentielle à résoudre peut se mettre sous la forme :

$$\frac{\mathrm{d}\boldsymbol{Y}(t)}{\mathrm{d}t} = \boldsymbol{F}(t, \boldsymbol{Y}(t)) \quad \text{avec} \quad \boldsymbol{F}(t, \boldsymbol{Y}(t)) = \begin{pmatrix} \frac{\mathrm{d}\omega_m(t)}{\mathrm{d}t} \\ K\omega_0^2 - \omega_0^2\omega_m(t) - 2m\omega_0\frac{\mathrm{d}\omega_m(t)}{\mathrm{d}t} \end{pmatrix}$$

La réponse $\omega_m(t)$ recherchée sur l'intervalle $[0, T_{\max}]$ sera obtenur par...la méthode d'Euler explicite ;-)
Le pas de temps, noté pas, sera choisi constant. L'intervalle de temps discrétisé est alors représenté par le tableau $T = [t_0 = 0, t_1, ..., t_{N-1} = T_{\max}]$.
Pour chaque pas de temps, une valeur approchée $\boldsymbol{Y}_i$ de la solution $\boldsymbol{Y}(t_i)$ de l'équation différentielle est recherchée. L'ensemble des $\boldsymbol{Y}_i$ représente N vecteurs de dimension 2, qui seront stockés en mémoire sous la forme du tableau :

$$\boldsymbol{SY} = \begin{pmatrix} \omega_m(0) & \omega_m'(0) \\ \omega_m(t_1) & \omega_m'(t_1) \\ \vdots & \vdots \\ \omega_m(T_{\max}) & \omega_m'(T_{\max}) \end{pmatrix}$$

Et voici enfin la première question...

1. Écrire un algorithme ou une fonction permettant de calculer le tableau T.
2. Écrire une fonction `f1(ti,yi)` qui prend en arguments la valeur du temps discrétisé i et la valeur du vecteur $\boldsymbol{Y}$ au temps discrétisé `ti` et qui retourne la valeur de $\boldsymbol{F}(t, \boldsymbol{Y}(t))$.
3. Donner la relation de récurrence qui lie $\boldsymbol{Y}_{i+1}$ à $\boldsymbol{Y}_i$ et à $\boldsymbol{F}(t, \boldsymbol{Y}(t))$ en fonction du pas de temps pas.

4. Écrire une fonction `EulerExplicite(Yini,h,Tmax,F)` qui prend en arguments `Yini`, un tableau de dimension 2 contenant la condition initiale de $\boldsymbol{Y}(t)$, `h` le pas de temps, `Tmax` l'instant final du calcul, et `F` la fonction du problème de CAUCHY.

Cette fonction renverra le tableau `SY`. L'appel à cette fonction dans le programme se fera avec la commande `SY = EulerExplicite(Y0,pas,tmax,f1)`.

5. Si le pas de temps es divisé par un facteur 10, comment évolue l'erreur de calcul ?

6. Donner la complexité de cette méthode pour $T_{\max}$ fixé et indiquer comment évolue le temps de calcul quand le pas de temps est divisé par un facteur 10.

On suppose que la quantité de mémoire nécessaire pour réaliser le calcul se limite au stockage de la matrice **SY** et du vecteur **T**. Ces éléments sont représentés en mémoire sous forme de tableaux de flottants en double précision.

7. Déterminer le nombre d'octets nécessaire en mémoire pour réaliser cette simulation numérique avec nombre de pas de temps $N = 10\,000$.

8. **Convergence de l'algorithme**. Avant d'appliquer l'algorithme de recherche du temps de réponse à 5%, il est nécessaire de savoir si l'algorithme de simulation converge. Après calcul, la solution ω_m est stockée dans la variable **W**.

On suppose que la solution converge si toutes les valeurs de ω_m pour t appartenant à $[0.9T_{\max}, T_{\max}]$ ne s'écartent pas de la valeur finale calculée, $\omega_m(T_{\max})$, de plus de $0,1\%$.

Écrire une fonction `TestConvergence(t,w)` qui prend en argument le tableau des temps `t` et la solution `w` et qui renvoie `True` si le critère est vérifié et `False` sinon.

9. **Détermination du temps de réponse à 5% du système**

Écrire une fonction `CalculT5` qui renvoie le temps de réponse à 5% si la convergence est vérifiée et -1 s'il n'est pas vérifié. Les entrées et sorties de cette fonction seront clairement définies.

La programmation et la simulation de l'algorithme précédent, dans les conditions de l'étude, ont permis de déterminer une temps de réponse de $0.045s$.

ANNEXE F

Dessine-moi une matrice...

Aujourd'hui, des mathématiques concrètes vont nous permettre de réduire, agrandir, assombrir, éclaircir, compresser, bruiter, quantifier,... une photo. Pour cela, il existe des méthodes provenant de la théorie du signal et des mathématiques continues. Nous nous pencherons plutôt sur des méthodes plus légères basées sur l'algèbre linéaire et l'analyse matricielle. Une image sera pour nous une matrice carrée de taille 2^9 à coefficients dans $[\![0, 2^9 - 1]\!]$. Cela manque de charme ? C'est sans compter sur Lena qui depuis quarante ans rend les maths sexy (la population hantant les laboratoires mathématiques et surtout informatiques est plutôt masculine ...).

Nous étudierons pour cela la « Décomposition en Valeurs Singulières » qui apparaît de nos jours comme un couteau suisse des problèmes linéaires : en traitement de l'image et de tout signal en général, en reconnaissance des formes, en robotique, en statistique, en étude du langage naturel, en géologie, en météorologie, en dynamique des structures, en....

Dans quel domaine travaille-t-on alors : algèbre linéaire, analyse, probabilités, topologie,... ? Un peu de tout cela et d'autres choses encore : cela s'appelle la mathématique.

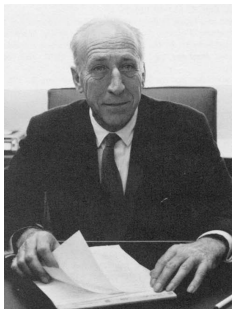
Lena

W. Kahan (né en 1933)

Nous allons travailler avec des images qui sont des matrices de niveaux de gris. Notre belle amie Léna sera représentée par une matrice carrée de taille 2^9 ce qui permet de reproduire Léna à l'aide de $2^{18} = 262\,144$ pixels. Léna prend alors beaucoup de place. Nous allons tenter de compresser la pauvre Léna sans pour cela qu'elle ne perde sa qualité graphique. Une des méthodes les plus abordables est d'utiliser la décomposition d'une matrice en valeurs singulières.
C'est un sujet extrêmement riche qui a de nombreuses applications. L'algorithme que nous utiliserons (mais que nous ne détaillerons pas) a été mis au point par deux très éminents chercheurs en 1965 (Gene Golub, états-unien et William Kahan, canadien, père de la norme IEEE-754). Il s'agit donc de mathématiques assez récentes, au moins en comparaison avec votre programme...
La petite histoire dit que des chercheurs américains de l'University of Southern California étaient pressés de trouver une image de taille 2^{18} pixels pour leur conférence. Passe alors un de leurs collègues avec, en bon informaticien, le dernier Playboy sous le bras. Ils décidèrent alors d'utiliser le poster central de la Playmate comme support...
La photo originale est ici : http://www.lenna.org/full/len_full.html mais nous n'utiliserons que la partie scannée par les chercheurs, de taille $5.12\text{in} \times 5.12\text{in}$...

La SVD

Le théorème



C. Eckart (1902-1973)

Un théorème démontré officiellement en 1936 par Carl Eckart et Gale Young affirme alors que toute matrice rectangulaire A se décompose sous la forme :

$$A = U \times S \times {}^tV$$

avec U et V des matrices orthogonales et S une matrice nulle partout sauf sur sa diagonale principale qui contient les valeurs singulières de A rangées dans l'ordre décroissant. Sylvester s'y était déjà intéressé pour des matrices carrées réelles en 1889 mais ce résultat a semblé n'intéresser personne pendant des années.
Le court article de Eckart et Young est disponible à l'adresse suivante :
http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.bams/1183501633.
Ce qui est remarquable, c'est que n'importe quelle matrice admet une telle décomposition, alors que la décomposition en valeurs propres (la diagonalisation d'une matrice) n'est pas toujours possible. Et quand on dit n'importe quelle matrice, on ne rigole pas : toute matrice, même rectangulaire. C'est vraiment très puissant et à la fois simple mais cette décomposition n'a trouvé d'applications importantes qu'assez récemment (ce qui veut dire après 1899 à l'échelle de votre programme de prépa...) ce qui explique peut-être qu'on en parle si peu en premier cycle.
Cependant, la décomposition en éléments singuliers utilise la décomposition en éléments propres.
Notons r le rang de A et U_i et V_i les vecteurs colonnes de U et V. La décomposition s'écrit :

$$A = \begin{pmatrix} U_1 & U_2 & \cdots & U_r & \cdots & U_m \end{pmatrix} \times \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_r & & \\ & & & \ddots & \\ & & & & 0 \end{pmatrix} \times \begin{pmatrix} {}^tV_1 \\ \vdots \\ {}^tV_r \\ \vdots \\ {}^tV_n \end{pmatrix}$$

ou formulé autrement :

$$\begin{aligned} A &= \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_r \cdot U_r \times {}^tV_r + 0 \cdot U_{r+1} \times {}^tV_{r+1} + \cdots \\ &= \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_r \cdot U_r \times {}^tV_r \end{aligned}$$

Il faut ensuite se souvenir que les σ_i sont classés dans l'ordre décroissant ce qui va avoir des applications importantes en informatique.

Interprétation géométrique

Comment interpréter géométriquement ce résultat ? Considérez une sphère de rayon 1 : que devient-elle transformée par A, i.e. par $U \times S \times {}^tV$? Faites un petit dessin en dimension 2. Que vaut $\|A\|_2 = \max_{\|x\|=1} \|Ax\|_2$?
Afin de répondre, un peu de vocabulaire d'abord. On a $AV_i = \sigma_i U_i$: on dit alors que U_i est un vecteur singulier à gauche pour la valeur singulière σ_i.

De même ${}^tAU_i = \sigma_i V_i$ et V_i est un vecteur singulier à droite pour la valeur singulière σ_i. Les bases (u_i) et (V_i) sont orthonormées : si l'on exprime la matrice de l'application linéaire associée à A dans ces bases, que pouvez-vous dire géométriquement ?

Un exemple simple

On suppose que la SVD d'une matrice A de taille $m \times n$ s'écrit $U \times S \times {}^tV$.

1. Quelles sont les dimensions de U et V ? Que pensez-vous de $A \times {}^tA$? Que représentent les colonnes de U pour $A \times {}^tA$?. Et les colonnes de V pour ${}^tA \times A$.
2. Calculez (à la main...) la SVD de $A = \begin{pmatrix} 2 & -2 \\ 1 & 1 \end{pmatrix}$.
3. Shoot again avec $\begin{pmatrix} 3 & 1 & 1 \\ -1 & 3 & 1 \end{pmatrix}$

Quels liens existent entre la SVD et la recherche des éléments propres d'une matrice ? Avec le théorème spectral ? Vaut-il mieux utiliser $A \times {}^tA$ ou ${}^tA \times A$?

Approximation de rang minimum d'une matrice

Dans de nombreux domaines, on doit travailler avec de grosses matrices. Il est alors très important de déterminer une matrice $\widetilde{A_s}$ de même taille qu'une matrice A mais de rang s inférieur au rang r de A et qui minimise l'erreur commise pour une certaine norme (qui est le plus souvent la norme de Frobenius, i.e. la racine carrée de la somme des carrés des coefficients). Le théorème d'Eckart-Young permet alors de dire que :

$$\min_{\mathrm{rg}(X)\leqslant s} \|A - X\|_F = \|A - \widetilde{A_s}\|_F = \sqrt{\sum_{j=s+1}^{r} \sigma_j^2(A)}$$

avec $\widetilde{A_s} = \sigma_1 \cdot U_1 \times {}^tV_1 + \sigma_2 \cdot U_2 \times {}^tV_2 + \cdots + \sigma_s \cdot U_s \times {}^tV_s$: on considère que les éléments diagonaux de A sont nuls en-dessous d'un certain seuil. Nous allons l'illustrer informatiquement à défaut de le démonter. On voit tout de suite des conséquences pratiques, que nous allons également illustrer informatiquement dans le cas de la compression des images.

Manipulation d'images

Quelques fonctions Python

Python contient une matrice carrée de taille 512 contenant les niveaux de gris représentant Lena. Il suffit de cherger les bonnes bibliothèques :

Python

```
from pylab import *
from scipy import misc

l = misc.lena()
imshow(l,cmap=cm.gray)
```

Lena est bien une matrice :

Python

```
In [8]: type(l)
Out[8]: numpy.ndarray

In [9]: shape(l)
Out[9]: (512, 512)
```

On peut préférer travailler sur une photo de format jpg (qui sera à l'envers) ou png (qui sera à l'endroit). Python la transforme en matrice avec la fonction imread. La matrice obtenue est en RVB : chaque coefficient est un vecteur (R,V,B). On la convertit en niveau de gris par exemple en calculant la moyenne des trois couleurs.
On récupère d'abord une image :

Python

```
In [4]: from urllib.request import urlretrieve
```

```
In [5]:
    urlretrieve('http://download.tuxfamily.org/tehessinmath/les_images/marty.png','marty.png')
Out[5]: ('marty.png', <http.client.HTTPMessage at 0x7f3aeeb5fa20>)
```

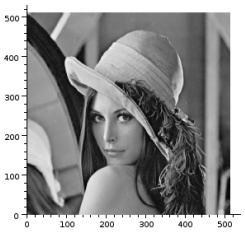
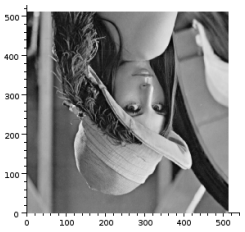
et on la transforme en matrice :

Python

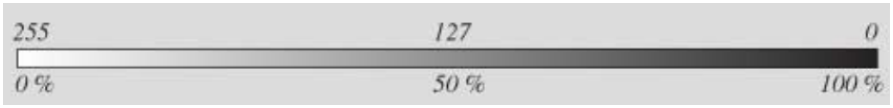
```python
tm = imread('marty.png')

def rvb_to_gray(m):
    r = len(m)
    c = len(m[0])
    return(array([[m[i,j][0] for j in range(c)] for i in range(r)]))

marty = rvb_to_gray(tm)
```

Manipulations basiques

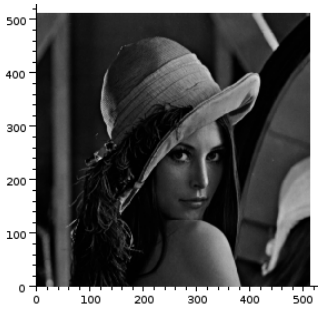
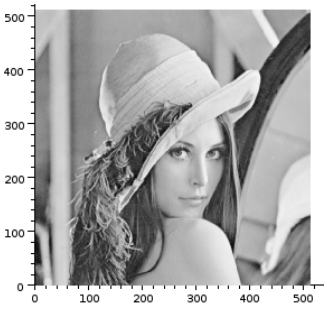
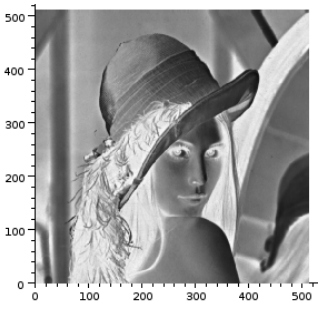
Comment obtenir les images suivantes :

Sachant que l'échelle de gris est celle-ci :



comment obtenir les images suivantes :

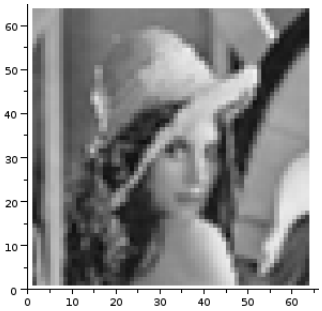
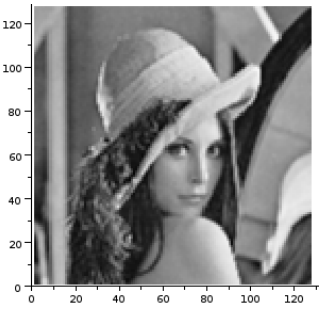
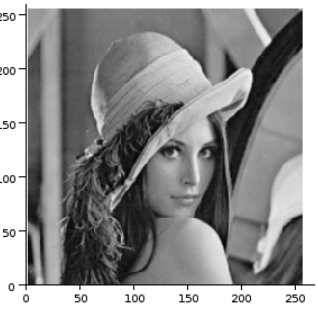




Gagner de la place

Une matrice de taille 2^9 contient 2^{18} entiers codés entre 0 et $2^8 - 1$ ce qui prend pas mal de place en mémoire. Peut-on être plus économique sans pour cela diminuer la qualité esthétique de la photo ?

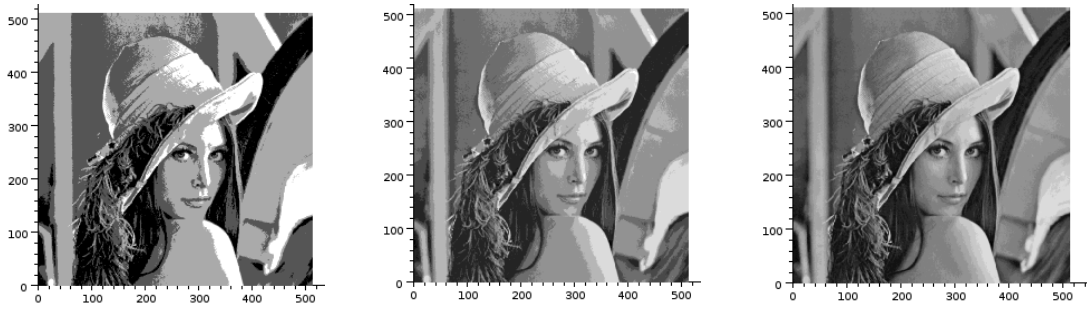
Résolution

La première idée est de prendre de plus gros pixels, c'est-à-dire une matrice plus petite : on extrait par exemple régulièrement un pixel sur k (en choisissant k parmi une puissance de 2 inférieure à 2^9).
Comment obtenir par exemple ces images ?

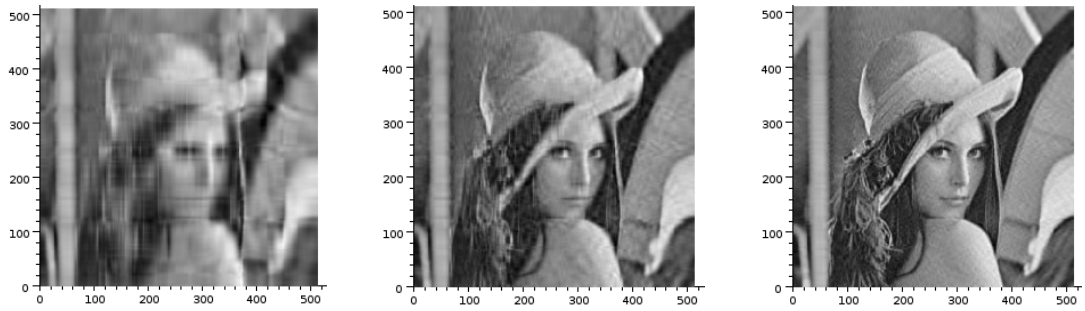
Quantification

On peut également réduire le nombre de niveaux de gris en regroupant par exemple tous les niveaux entre 0 et 63 en un seul niveau 0, puis 64 à 127 en 64, 128 à 191 en 128, 192 à 256 en 192.
Par exemple, avec 4, 8 puis 16 niveaux :

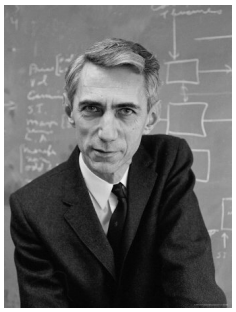


Compression et SVD

Tout langage orienté calcul comprend des fonctions liées à la décomposition en éléments simples.
Python a sa fonction `linalg.svd(mat)` qui renvoie la décomposition en valeurs singulières sous la forme du triplet U, S et V.
En vous inspirant des commentaires faits dans les paragraphes précédents sur la SVD, expliquez comment cette décomposition permet de compresser une image. Quantifiez également le niveau de compression.
Voici trois niveaux de compression :



Enlever le bruit



C.E. Shannon
(1916-2001)

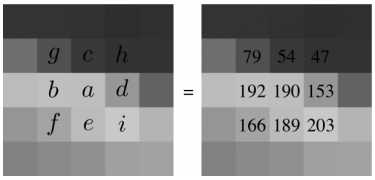
La transmission d'informations a toujours posé des problèmes : voleurs de grands chemins, poteaux télégraphiques sciés, attaques d'indiens, etc.
Claude Elwood Shannon (1916 - 2001) est un mathématicien-inventeur-jongleur américain qui, suite à son article « *A mathematical theory of communications* » paru en 1948, est considéré comme le fondateur de la *théorie de l'information* qui est bien sûr une des bases de...l'*informatique*.
L'idée est d'étudier et de quantifier l'« information » émise et reçue : quelle est la compression maximale de données digitales ? Quel débit choisir pour transmettre un message dans un canal « bruité » ? Quel est le niveau de sûreté d'un chiffrement ?...
La théorie de l'information de Shannon est fondée sur des modèles probabilistes : leur étude est donc un préalable à l'étude de problèmes de réseaux, d'intelligence artificielle, de systèmes complexes.
Par exemple, dans le schéma de communication présenté par Shannon, la source et le destinataire d'une information étant séparés, des perturbations peuvent créer une différence entre le message émis et le message reçu. Ces perturbations (bruit de fond thermique ou accoustique, erreurs d'écriture ou de lecture, etc.) sont de nature *aléatoire* : il n'est pas possible de prévoir leur effet. De plus, le message source est par nature *imprévisible* du point de vue du destinataire (sinon, à quoi bon le transmettre).
Shannon a également emprunté à la physique la notion d'entropie pour mesurer le désordre de cette transmission d'information, cette même notion d'entropie qui a inspiré notre héros national, Cédric Villani...
Mais revenons à Lena. Nous allons simuler une Lena bruitée :

Python

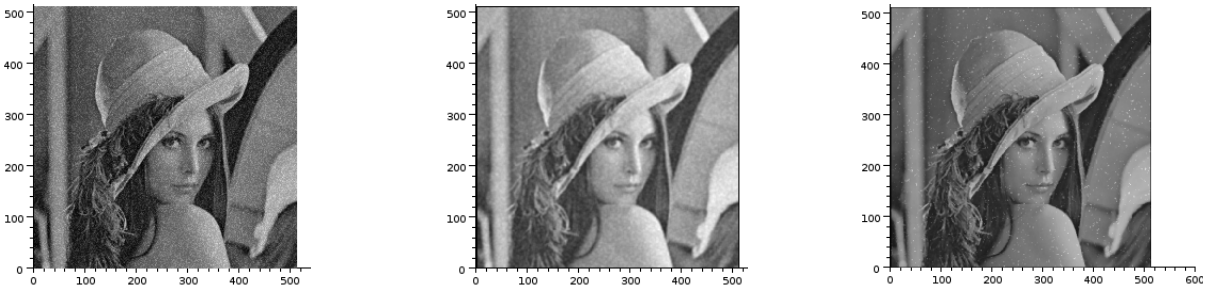
```
from random import *
def ber(p):
    if random() < p:
        return 1
    else:
        return 0

def bruit(m,p):
    r = len(m)
```

```
    c = len(m[0])
    return(array([[(m[i,j] + randint(0,255)*ber(p))%256 for j in range(c)] for i in
        range(r)]))
```

Il existe de nombreuses méthodes, certaines très élaborées, permettant de minimiser ce bruit. Une des plus simples est de remplacer chaque pixel par la moyenne de lui-même et de ses 8 autres voisins directs, voire aussi ses 24 voisins directs et indirect, voire plus, ou de la médiane de ces séries de « cercles » concentriques de pixels, comme on le voit sur cette figure (extraite du site Images des mathématiques) :



Créez des fonctions qui feront le boulot. Vous utiliserez mean et median pour obtenir par exemple :



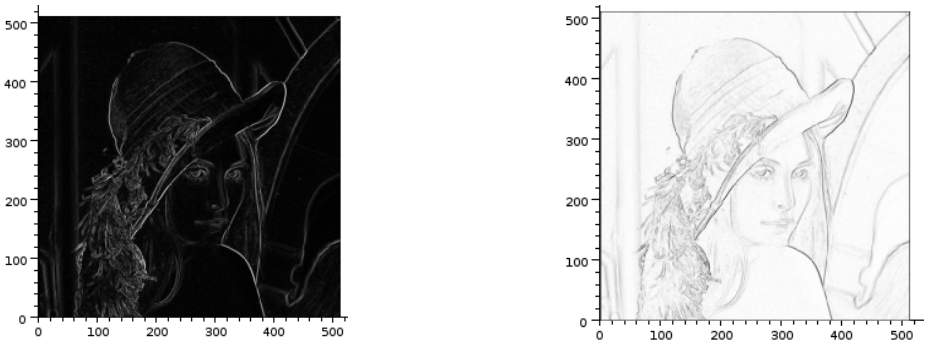
où l'on trouve de gauche à droite l'image bruitée originale puis sa correction par moyenne sur 25 pixels et par médiane sur 9 pixels.

Détection de bords

Pour sélectionner des objets sur une image, on peut essayer de détecter leurs bords, i.e. des zones de brusque changement de niveaux de gris.
On remplace alors un pixel par une mesure des écarts des voisins immédiats donnée par exemple par :

$$\sqrt{(m_{i,j+1}-m_{i,j-1})^2+(m_{i+1,j}-m_{i-1,j})^2}$$

On obtient alors au choix :



LECTURES RECOMMANDÉES POUR ALLER PLUS LOIN

H. Abelson, G. Sussman et J. Sussman,
Structure and interpretation of computer programs :,
vol. MIT electrical engineering and computer science series, MIT Press, ISBN 0-262-51087-1, 1996.

A. Benoit, Y. Robert et F. Vivien,
A Guide to Algorithm Design : Paradigms, Methods, and Complexity Analysis,
vol. Applied Algorithms and Data Structures series, Chapman & Hall/CRC, août 2013.

A. Brygoo, T. Durand, M. Pelletier, C. Queinnec et M. Soria,
Programmation récursive (en Scheme) - Cours et exercices corrigés,
vol. Informatique, Dunod, ISBN 9782100528165, 2004.

T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein,
Introduction to Algorithms, Third Edition,
The MIT Press, 3rd édition, ISBN 0262033844, 9780262033848, 2009.

E. W. Dijkstra,
« Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60 »,
cwireport MR 34/61, Stichting Mathematisch Centrum, adresse : http://oai.cwi.nl/oai/asset/9251/9251A.pdf, 1961,
(ALGOL Bulletin, 1).

J. Dufourd, D. Bechmann et Y. Bertrand,
Spécifications algébriques, algorithmique et programmation,
vol. I.I.A. Informatique intelligence artificielle, InterEditions, ISBN 9782729605810, 1995.

D. E. Knuth,
The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms,
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, ISBN 0-201-89683-4, 1997.

D. E. Knuth,
The Art of Computer Programming, Volume 3 : (2Nd Ed.) Sorting and Searching,
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, ISBN 0-201-89685-0, 1998.

G. J. E. Rawlins,
Compared to What ? An Introduction to the Analysis of Algorithms,
Computer Science Press, 1992.

R. Sedgewick et P. Flajolet,
An Introduction to the Analysis of Algorithms,
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0-201-40009-X, 1996.

R. Sedgewick et K. Wayne,
Algorithms, 4th Edition.,
Addison-Wesley, ISBN 978-0-321-57351-3, 2011.