

Algorithmique et programmation

Philippe Rannou

ESIR

Organisation du module

Volume horaire

44 H (sur 12 semaines) :

- sur 2 semaines \approx 2H de CM + 2H de TD + 4H de TP

Modalités d'évaluation

- TP(s) à rendre
- 1 DS (milieu de semestre)
- 1 DS (fin de semestre)

Équipe pédagogique


- CM : Philippe Rannou
- TD : Philippe Rannou et Hélène Feuillâtre
- TP : Philippe Rannou et Hélène Feuillâtre

Contenu

- Base des concepts algorithmiques (variables, conditionnelles, boucles)
- Syntaxe élémentaire du langage python
- Structures de données composées (tableaux et dictionnaires)
- Notion de récursivité

Compétences visées

- Concevoir, analyser et modifier des algorithmes simples
- Décomposer un problème en sous-problèmes
- Implémenter ces algorithmes en python
- Tester et déboguer des programmes
- Déterminer la complexité d'un algorithme

les exercices qui jalonnent ce cours sont signalés par un : , ils seront corrigés en cours et serviront en TDs/TPs.

Plan du cours

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Quelques exemples d'algorithmes de la vie courante

Recette de cuisine

- *Éplucher et couper les potirons.*
- *Émincer l'ail et l'oignon.*
- *Faire suer l'oignon dans l'huile d'olive.*
- ...

Notice d'utilisation

- *Veillez à ce que les tuyaux soient bien connectés.*
- *Branchez votre machine.*
- *Ouvrez complètement le robinet.*
- ...

Instructions GPS

- *Prendre N137 à Pont-Péan et quitter D286.*
- *Continuer sur N137.*
- *Prendre la sortie en direction de Caen.*
- ...

Qu'est-ce qu'un algorithme ?

Ces exemples ont des caractéristiques communes :

- ce sont des suites d'instructions élémentaires,
- ils permettent de résoudre des tâches déterminées.

Définition 0.1

*Un **algorithme** est une suite finie d'instructions et permettant de résoudre un problème.*

*Le domaine qui étudie les algorithmes est appelé **l'algorithmique**.*

Dans ce cours, nous étudierons l'algorithmique dans le cadre de l'*informatique*.

Informatique

L'informatique est la science du traitement automatique de l'information.

Elle englobe :

- l'étude des programmes immatériels qui décrivent un traitement à réaliser (logiciel, *software*).
- l'étude des machines qui exécute ce traitement (matériel, *hardware*)
- L'algorithmique fait partie du côté *logiciel* de l'informatique.

Algorithme vs Programme Informatique

- En TP vous *implémenterez* des algorithmes sur ordinateur, c'est à dire que vous *traduirez* ces algorithmes en **programmes informatiques** (ici en langage python)
- On pourrait donc utiliser 2 langages dans ce cours :
 - du **pseudo-code** pour écrire les algorithmes,
 - le langage python pour écrire les programmes.

Algorithme 1 : EstPositif(x)

Entrées : x : entier

```
1 début
2   si  $x \geq 0$  alors
3     | Afficher "Oui"
4   sinon
5     | Afficher "Non"
6   fin
7 fin
```

Pseudo-code

```
def est_positif(x) :
    if x >= 0 :
        print("Oui")
    else :
        print("Non")
```

python

⇒ Dans ce cours on n'écrira
que du python

- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Formalisation

En informatique, les algorithmes sont des suites d'**instructions** qui manipulent des **données**.

Données

Les **données** peuvent être de différents types :

- entier, booléen, réel, chaîne de caractères, ...
- images, vidéos, ...
- fichiers, ...
- ...

→ Toutes ces données sont codées en binaires (avec des "0" et des "1")

Instructions

Les **instructions** utilisées en algorithmique (if, while, for, ...) correspondent aux **briques de base** que peut faire directement l'ordinateur.

- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Définition 1.1

*Toute donnée est soit une **variable** soit une **constante**:*

- *une **variable** est une donnée dont la valeur va évoluer lors de l'exécution du programme*
- *une **constante** va conserver la même valeur tout au long du traitement*

Chaque donnée (variable comme constante) est définie par :

- un **identificateur**
- un **type**
- une **valeur**

Programmation

- Dans certains langage, comme le C, Java ou C++, la gestion des variables et constantes est différenciée.
- En python, toutes les données que l'on manipule sont des variables.

Définition 1.2

L'**identificateur** correspond au nom que l'on donne à une donnée pour l'identifier (variable comme constante) :

- composé à partir des lettres (non accentuées) de l'alphabet, des chiffres et du symbole underscore '_' (les autres symboles comme l'espace ou le tiret '-' ne sont pas autorisés)

Conventions pour les identificateurs

Il existe plusieurs conventions de nommage suivants les langages :

- `nombredejours`
- `nombre_de_jours` : pour les variables/fonctions en python
- `NombreDeJours`
- `NOMBREDEJOURS`
- `NOMBRE_DE_JOURS` : pour les constantes en python
- ...

Noms à éviter

- `l` ("L" minuscule)
- `I` ("i" majuscule)
- `O` ("o" majuscule : trop proche du zéro)

Pour plus d'informations sur les conventions de nommage en python :

<https://www.python.org/dev/peps/pep-0008/>

Définition 1.3

Le **type** d'une donnée (ou plus simple **type**) désigne les **valeurs** que peut prendre une donnée ainsi que les **opérateurs** qui lui sont applicables.

- Les valeurs que peuvent prendre une donnée sont codées en binaire
- À la nature (numérique, caractère ...) de l'information mémorisée correspond généralement une manière de coder cette information.
- Le type d'une variable permet d'associer entre eux : la nature des informations, le codage mais aussi les limites et opérations associées.

Types de données en python

- Booléen (`bool`) :
 - valeur : `True` ou `False` (Attention aux majuscules)
- Entier (`int`)
 - valeur : n'importe quelle valeur entière
 - exemple : 18, -3, +326
- Réel (`float`)
 - valeur : n'importe quelle valeur réelle
 - exemple : 1.0, 3.14, -128.32
- Chaîne de caractères (`string`) :
 - valeur : ensemble de lettres, chiffres, symboles ...
 - exemple : "ABCD" , "a"

Déclaration

Dans certains langages (C, Java, C++), le typage est **explicite**, c'est à dire que le type est précisé lors de la déclaration. (Ex : `int a = 5;`)
En python, la plupart du temps, on ne précise pas le type des variables que l'on manipule.

Définition 1.4 (Affectation)

Une **affectation** *identifiant* = *expression* est une instruction qui permet de spécifier qu'au moment de son exécution, la variable désignée par *identifiant* recevra comme nouvelle valeur le résultat de *expression*.

En python, l'affectation s'effectue avec le signe "=", et le type d'une variable peut changer :

```
n = 8  
n = "ABC"
```

→ **MAIS !** C'est fortement déconseillé ! (et souvent impossible dans les autres langages)

Affectation (2/2)

`identifiant = expression`

- Une expression permet de désigner le calcul d'une nouvelle valeur à partir d'autres valeurs et d'opérations.
- Les valeurs utilisées dans l'expression peuvent être des constantes, des valeurs de variables, des données littérales ...:
 - $3 + 27$ ou $x * 3,0$ si x est une variable de type réel
- Lors de l'affectation `identifiant = expression` :
 - 1 L'expression est d'abord évaluée.
 - 2 Le résultat de l'expression (donc la valeur) est ensuite affectée à la variable désignée par son identifiant.

→ on peut donc utiliser la valeur d'une variable dans sa propre affectation :

$$x = x + 3$$

Conversion de type en python

- En python, il est parfois possible de convertir une expression d'un type en un autre.
- Pour ce faire, on écrit : `type_d_arrivee(expression)`.

Exemples :

- `float(3)` → 3.0
- `int(5.6)` → 5
- `string(68)` → "68"
- `int("35")` → 35
- `float("08.93")` → 8.93
- `str(09.64)` → "9.64"

Définition 1.5

*Un **opérateur** est un symbole indiquant une opération.*

- Les opérateurs sont souvent binaires (c'est-à-dire reliant **deux** opérandes) mais peuvent aussi être unaires (une seule opérande), ternaires voire n -aires.
- Un opérateur est généralement associé à un type de données. (le "+" a la même signification lorsqu'il est utilisé sur des entiers ou des réels, mais pas sur des chaînes de caractères).
- Trois familles d'opérateurs sont distinguées :
 - ① opérateurs arithmétiques : donnent un résultat numérique à partir d'opérandes numériques (addition, soustraction ...)
 - ② opérateurs relationnels : donnent un résultat logique (booléen) à partir d'opérandes numériques (plus grand que, égal ...)
 - ③ opérateurs logiques : donnent un résultat logique à partir d'opérandes logiques (et logique ...)

Opérateurs sur les chaînes de caractères

Une variable de type chaîne de caractères :

- ne peut utiliser d'opérateurs arithmétiques
- peut utiliser les opérateurs relationnels : $>$, $>=$, $<$, $<=$, $=$ et $!=$ (utilisation de l'ordre *lexicographique*, i.e. celui du dictionnaire)
- possède un opérateur de concaténation : $\ll + \gg$

Exemple : "abc" + "bcd" retourne "abcbcd"

Remarque

En python, on peut effectuer l'opération `chaine*k`, avec `k` entier : cela donnera le résultat `chaine chaine ... chaine` où `chaine` est répétée `k` fois.

- En python, on utilise `True` et `False` pour désigner les deux valeurs booléenne.
- Les seuls opérateurs utilisables avec le type booléen sont les opérateurs logiques ("`et`" : `and`, "`ou`" : `or`, "`non`" : `not`) et les opérateurs relationnels `==` et `!=`
- Un petit rappel de logique :
 - l'expression `a or b` est vraie si l'une des variables `a` ou `b` est vraie.
 - l'expression `a and b` est vraie si les deux variables `a` et `b` sont vraies.
 - l'expression `not a` est le contraire de `a`.

Opérateurs sur les entiers

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $//$ (division entière) et $\%$ (modulo : reste de la division entière) sont disponibles sur les entiers et donnent toujours un entier (les opérateurs relationnels sont aussi disponibles) :

```
x = 13 // 2
```

```
# la valeur de x est donc 6
```

```
# (ici // désigne la division euclidienne)
```

```
x = 13 % 2
```

```
# la valeur de x est donc 1
```

```
# (ici % désigne le reste de la division)
```

Remarques :

- Dans certains langages (C, Java, C++), une variable de type entier ne peut prendre qu'un nombre fini de valeurs qui est fonction du nombre d'octets nécessaires à son codage. Un entier qui serait codé sur 1 octet (8 bits) ne pourrait prendre que 256 valeurs (2^8) de 0 à 255 par exemple. En python il n'y a pas de limite.

Opérateurs sur les réels

- Les opérations arithmétiques binaires telles que $+$, $-$, $*$, $/$ sont disponibles sur les réels et donnent toujours un réel.
- Les opérateurs relationnels sont aussi disponibles sur les réels **mais** les opérateurs `==` et `!=` sont **absolument** à éviter !

Remarques :

- Une variable de type réel permet la représentation des nombres avec une certaine précision qui est fonction du nombre d'octets utilisées pour son codage.
- L'opération `/` est aussi disponible sur les entiers en python, mais elle renvoie systématiquement un réel (même si le résultat est entier).



Exercice

Algorithme	Type de la variable ?	Valeur ?
$tva = 19.6$		
$x = 4/3$		
$n = 17\%5$		
$m = 28 // 3$		



Exercice

Algorithme	Type de la variable ?	Valeur ?
<pre>b = "True"</pre>		
<pre>c = True and (False or True)</pre>		
<pre>mot = "Le"+" chat"</pre>		
<pre>mot2 = "3+3=" +6</pre>		

- 1 Concepts de bases
 - Types de données
 - **Entrées-sorties**
 - Structures de contrôle
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

- Un programme a généralement besoin de communiquer avec l'extérieur :
 - affichage de résultats sur un écran,
 - demande à l'utilisateur de fournir une donnée,
 - ...
- Dans la réalité les instructions d'écriture et de lecture se font sur des périphériques :
 - la lecture spécifie qu'une nouvelle valeur d'une variable doit être lue sur un périphérique (typiquement à l'aide du clavier)
 - l'écriture permet d'écrire sur un périphérique (typiquement l'écran) la valeur d'une variable ou d'une expression.

Définition 1.6

La **lecture** est une instruction d'affectation à une variable d'une valeur saisie (au clavier) lorsque l'algorithme est exécuté.

En python:

```
x = input()
```

Attention !

- En python, le type par défaut d'une lecture sera une chaîne de caractère.
- Si l'on souhaite, par exemple, un entier, il faut écrire :

```
x = int(input())
```

Définition 1.7

*L'**écriture** est une instruction permettant d'écrire (c'est-à-dire d'afficher à l'écran) les valeurs des résultats ainsi que d'éventuels commentaires appropriés.*

En python:

```
x = 3  
print(x)
```

- Pour afficher plusieurs chaînes `c1`, `c2`, `c3`, on peut utiliser la commande `print(c1+c2+c3)`. **Mais cela ne fonctionne qu'avec des chaînes de caractères.**
- Pour afficher plusieurs arguments de type différents `a1`, `a2`, `a3`, on peut utiliser la commande `print(a1,a2,a3)`.



Exercice

Qu'affiche l'algorithme suivant à l'écran ?

```
print("Entrez une valeur : ")  
x = input()  
y = 2*x  
x = 2*x+y  
print(x)
```

Cet algorithme :



- 1 Concepts de bases
 - Types de données
 - Entrées-sorties
 - Structures de contrôle
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité



Exemple

Écrivez un algorithme demandant à l'utilisateur de rentrer un entier et affichant la valeur absolue de cet entier.

- Jusqu'à maintenant, les instructions d'un programme s'exécutaient dans l'ordre de leur apparition.
- Nous avons aussi besoin, dans un algorithme, de pouvoir :
 - faire un choix (qui dépend par exemple d'une saisie de l'utilisateur),
 - répéter plusieurs fois une même série d'instructions.

Définition 1.8

*Une **structure de contrôle** est une instruction destinée à commander le déroulement du programme.*

Structure conditionnelle (1/2)

Définition 1.9

Une **structure conditionnelle** permet de réaliser une série d'instructions plutôt qu'une autre (donc de faire un choix) en fonction du résultat d'un test (donc d'une condition).

```
if condition :  
•   instructions 1
```

```
if condition :  
    instructions 1  
• else :  
    instructions 2
```

Si le résultat de "condition" est True alors l'algorithme réalise "instructions 1".

Si le résultat de "condition" est True alors l'algorithme réalise "instructions 1", sinon "instructions 2".

Structure conditionnelle (2/2)

`condition` est une expression dont l'évaluation retourne une valeur booléenne (donc `True` ou `False`)

Exemple :

```
if age < 18 :  
    print("personne mineure")  
else :  
    print("personne majeure")
```

Précisions syntaxiques en python

En python, on peut utiliser :

- une conditionnelle simple :

```
if condition :  
    instructions
```

Remarque : il faut une tabulation avant chacune des "instructions"

- une conditionnelle avec sinon :

```
if condition :  
    instructions1  
else :  
    instructions2
```

Remarque : le else est au même niveau que le if

- une conditionnelle avec succession de conditions :

```
if condition1 :  
    instructions1  
elif condition2 :  
    instructions2  
...
```



Exemple

Écrivez un algorithme qui affiche 3 fois la chaîne "coucou".

Écrivez un algorithme qui demande à l'utilisateur d'entrer un entier positif et affiche autant de fois la chaîne "coucou".

Définition 1.10

Une **structure répétitive** (ou *boucle*) permet de répéter plusieurs fois une série d'instructions en fonction d'une condition.

Il existe deux familles de structures répétitives, selon que l'on connaisse ou non à l'avance le nombre de répétitions à réaliser.

Exemples :

- si je dois faire la somme de dix entiers saisies au clavier alors je connais à l'avance le nombre de répétitions à réaliser.
- si je dois sortir du programme lorsque l'utilisateur saisie le caractère q (et uniquement dans ce cas) alors je ne connais pas à l'avance le nombre de répétitions.

Définition 1.11

*Une structure répétitive de type **while** permet de répéter une série d'instructions tant qu'une condition est vraie.*

```
while condition :  
    instructions
```

- *condition* s'appelle **la condition d'arrêt**.
- tant que la condition d'arrêt vaut True, le programme exécute "instructions"
- normalement, "instructions" finira par modifier la valeur de *condition*, sinon l'algorithme tournera sans fin.

Dans ce type de structure :

- le nombre d'itérations dépend de la valeur d'une condition
- ⇒ le nombre de répétitions n'est donc pas nécessairement connu à l'avance

Précisions syntaxiques en python:

En python, la boucle Tant que s'écrit `while` :

```
while condition :  
    instructions
```

Notez :

- les " :" après la condition
- le décalage (avec tabulation) des instructions.



Exemple while

Que fait cet algorithme ?

```
somme = 0
nombre = input()
while nombre != -1 :
    somme = somme + nombre
    nombre = input()
print(somme)
```

Structure Pour (1/2)

Définition 1.12

*Une structure répétitive de type **pour** permet de répéter un nombre de fois connu à l'avance une série d'instructions.*

```
for variable in ensemble_valeur :  
    instruction
```

Remarques :

- `variable` est une variable locale à la boucle, elle ne doit pas être modifiée par la série d'instructions.
- à chaque tour de boucle, `variable` va varier en prenant dans l'ordre les valeurs de `ensemble_valeurs`.

Structure Pour (2/2)

Exemple :

```
for i in [3, 5, 10] :  
    print(i)
```

- i va prendre successivement les valeurs 3, 5 et 10 : il y a aura trois itérations de cette boucle.

Il est possible de spécifier qu'une variable prenne toutes les valeurs dans une plage d'entiers donnée :

```
for i in range(deb, fin) :  
    instructions
```

Attention !

En python, la borne de fin est **exclude** ! i prendra donc les valeurs deb, deb+1, ..., fin-1.



Exemple Structure Pour

Qu'affiche l'algorithme suivant ?

```
somme = 0
```

```
for i in range (1,11) :
```

```
    somme = somme+i
```

```
print (somme)
```

Lien entre structure Tant que et Pour

Il est facile de transformer une boucle pour en une boucle tant que :

```
for i in range(1,10) :  
    instructions
```

→

```
i = 1  
while i < 10 :  
    instructions  
    i = i+1
```

L'inverse est loin d'être aussi simple et surtout « propre » :

```
i = 1  
j = 0  
while i <= 10 and j < i*2 :  
    instructions  
    i = i+1  
    j = i+j
```

→

```
j = 0  
for i in range(1,11) :  
    instructions  
    j = i+j  
    if j >= i*2 :  
        i = 11
```



Exercice

Écrire deux programmes permettant d'afficher les tables de multiplications de 1 à 10, l'un utilisant des boucles `pour`, l'autre utilisant des boucles `tant que`.

- 1 Concepts de bases
- 2 Structuration d'un programme**
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Structuration d'un programme

Motivation

- Lorsque la taille d'un programme augmente, sa **lisibilité** devient de plus en plus difficile.
- En outre, si certaines parties du programme sont répétées cela rend plus complexe la **maintenance** et cela nuit à la **réutilisabilité** de votre code.

Objectif

Rendre vos programmes concis, clairs, compréhensifs, maintenables et réutilisables

Solution

Décomposer un programme en **sous-programmes**

Exemple

```
# Volume d'un cylindre
r = 3
h = 2.3
v = 3.14 * r * r * h
print("Le volume du
      cylindre vaut", v)
```

```
def aire_disque(r) :
    return 3.14*r*r
```

```
# Volume d'un cylindre
→ r = 3
h = 2.3
a = aire_disque(r)
v = a * h
print("Le volume du
      cylindre vaut", v)
```

Définition 2.1

Un sous programme est une série d'instructions réalisant des traitements en fonction de données :

- *les données sont appelées arguments ou paramètres du sous-programme,*
- *le résultat du traitement est appelé valeur de retour du sous-programme.*

Avant d'être utilisé (c'est-à-dire appelé) un sous programme doit être déclaré.

Déclaration d'un sous-programme (1/2)

Définition 2.2

La déclaration d'un sous-programme comporte :

- une **signature** qui indique :
 - ① le nom du sous-programme
 - ② le nom (local) des paramètres à fournir
Elle peut éventuellement indiquer (en commentaire) :
 - ③ le rôle du sous-programme
 - ④ les pré-conditions : conditions que doivent remplir les données avant le début de l'algorithme
 - ⑤ les post-conditions : conditions que doivent remplir les résultats après réalisation de l'algorithme
- un **corps** qui indique la description des instructions à réaliser

Remarque

Les paramètres d'un sous-programmes sont représentés par des noms quelconques appelés **paramètres formels**.

Exemple déclaration sous-programme (python)

```
def perimetre(rayon) :  
    """  
        Fonction calculant le périmètre  
        d'un cercle à partir de son rayon  
  
        Parameters :  
            rayon (float) : doit être positif  
  
        Returns :  
            float : le périmètre du cercle  
    """  
  
    p = 2 * 3.14 * rayon  
    return p
```

- Si un sous-programme n'a pas besoin de retourner une valeur alors:
 - nous omettons l'instruction `return`
- Si un sous-programme n'a pas de paramètre alors nous laisserons les parenthèses vides

Exemple

```
def afficher_bonjour() :  
    print(" Bonjour !")
```

Définition 2.3

*L'appel d'un sous-programme est l'**invocation** du sous-programme par son nom en fournissant des données en nombre requis.*

- Les données fournies sont appelées **paramètres effectifs**.
- Les paramètres effectifs peuvent être : des constantes, des valeurs littérales, des expressions, des valeurs de variables, ...
- Dans le cas où un paramètre effectif est une variable :
 - la variable doit être initialisée
 - le sous-programme ne peut que lire la variable (donc il ne pourra pas modifier sa valeur)

Exemple

```
def p_cercle(rayon) :  
    p = 2*3.14*rayon  
  
print("Quel est le rayon ?")  
input(r)  
peri = p_cercle(r)  
print("Le périmètre est :", peri)
```

Notez :

- appel du sous-programme `p_cercle` avec passage du paramètre effectif *r* et récupération du résultat dans *peri*
- la valeur de retour (*p*) est retournée à l'instruction qui a appelée le sous-programme. (ici l'affectation à *peri*)

Définition 2.4

La portée d'une donnée désigne son niveau de visibilité :

- *Les données (variables, constantes et paramètres formels) déclarées dans un sous programme ont une **portée locale** à ce sous programme (c'est-à-dire que la donnée n'est pas visible en dehors du sous-programme).*
- *Les autres données auront une **portée globale** (c'est-à-dire que la donnée est visible à n'importe quel endroit du programme).*

Un bon algorithme **ne doit jamais** utiliser de variables globales dans un sous-programme. *Il peut en revanche utiliser des constantes.*



Exercices

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

```
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

```
a = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(a)
```

Qu'affiche l'algorithme ci-dessus ?

```
b = 4  
def g(a) :  
    b = a*a  
    return b
```

```
c = g(5)  
print(b)
```

Qu'affiche l'algorithme ci-dessus ?

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 **Données structurées**
 - Tableaux
 - Dictionnaires
 - Autres types de données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Définition 3.1

Un type de données est dit :

- **scalaire** lorsqu'il permet de stocker une seule information (on parle aussi de type atomique) : entiers, réels, caractères et booléens.
- **structuré** lorsqu'il permet de stocker plusieurs informations (on parle alors de type composite) : chaînes de caractères, tableaux, dictionnaires.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
 - Tableaux
 - Dictionnaires
 - Autres types de données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Définition 3.2

Un **tableau** est une structure de données permettant de représenter une collection d'éléments de même type :

- les éléments sont rangés consécutivement ;
- les éléments sont accessibles par un numéro d'ordre : leur **indice**.

En python

- Affectation : `tableau = [1,3,5,2,-3]`
- Accès (lecture et écriture) : `tableau[i]` avec `i` entier compris entre 0 et `len(tableau)-1`

En python, la taille des tableaux est modifiable. C'est une particularité : dans la plupart des autres langages, la taille des tableaux est fixe.

Exemple

```
def somme_des_element(tab) :  
    s = 0  
    for i in [2,5,6] :  
        s = s + tab[i]  
    return s
```

Compléments de commande en python 1/3

- $t_1 + t_2$: concaténation de t_1 et t_2 , mis bout-à-bout.

Exemple :

```
>>> [3,5] + [2,7,5]  
[3,5,2,7,5]
```

- $n * t$: concaténation de n occurrences de t ($t + \dots + t$)

Exemple :

```
>>> 3*[1,4]  
[1,4,1,4,1,4]
```

```
>>> 7*[0]  
[0,0,0,0,0,0,0]
```


Compléments de commande en python 2/3

- `t[a:b]` : tableau des valeurs de `t` indicées de `a` à `b-1`
- `t[a:]` : tableau des valeurs de `t` indicées de `a` à la fin
- `t[:b]` : tableau des valeurs de `t` indicées de 0 à `b-1`

Exemple :

```
>>> x = [-3,-2,-1,0,1,2,3,4,5,6,7]
>>> y = x[2:5] # clone le segment de l'indice 2 à 4
>>> y
[-1,0,1]

>>> y[1] = 18
>>> y
[-1,18,1]

>>> x
[-3,-2,-1,0,1,2,3,4,5,6,7] # x n'est pas modifié
```

Compléments de commande en python 3/3

- `in` : teste l'appartenance à un tableau

```
>>> 2 in [4,2,8]
True
```

- `del t[a]` : supprime l'élément d'indice `a` du tableau `t` >>> `x =`

```
[18,7,3,5,1,0,-2,9]
>>> del x[2]          # suppression de l'élément d'indice 2
>>> x
[18,7,5,1,0,-2,9]
```

- `del t[a:b]` : supprime les éléments d'indices `a` à `b-1` du tableau `t`

```
>>> del x[4:]          # suppression des 3 derniers éléments
>>> x
[18,7,5,1]
```

Définition 3.3

Une **méthode** (en *python*) est une fonction qui s'applique à un objet.

Syntaxe d'un appel : *nom_objet.nom_méthode(paramètres)*

- **append** : ajoute un élément à la fin

```
>>> a = [2,5,7,3]
>>> a.append (12)
>>> a
[2,5,7,3,12]
```

- **pop** : supprime le dernier élément et le renvoie

```
>>> a.pop ()
12
>>> a
[2,5,7,3]
```

Méthodes sur les tableaux : compléments

- `reverse` : renverse l'ordre des éléments
- `extend(t)` : rajoute les éléments du tableau `t` à la fin du tableau courant
- `count(e)` : compte le nombre d'occurrences de `e` dans le tableau
- `index(e)` : renvoie l'indice de la première occurrence de `e` dans le tableau
- `insert(i,e)` : insert `e` à l'indice `i`
- `sort` : trie par ordre croissant

Parcours de tableau : for

for x in t : crée une boucle dans laquelle x prend chaque valeur du tableau t

```
t = [1,5,2]
for x in t :
    print (x)
```

Affiche

1
5
2

Ex : Compter le nombre de valeurs paires dans la liste 1

```
def compte_pair(t) :
    compteur = 0
    for k in t :
        if k%2 == 0 :
            compteur+=1
    return compteur
```

```
def compte_pair(t) :
    compteur = 0
    n = len(t)
    for i in range(n) :
        if t[i]%2 == 0 :
            compteur+=1
    return compteur
```

Remarque : list(range (n)) permet de créer un tableau contenant tous les éléments du range (attention, range ne crée pas un tableau)

Parcours de tableaux : compléments

`for (i,e) in enumerate (t) :` parcourt le tableau `t` en mémorisant dans `i` les indices successifs et dans `e` les éléments successifs.

```
t = [1,5,2]
```

```
for (i,e) in enumerate (t) :
```

```
    print ("L'element d'indice ", i, " du tableau est ", e)
```

Affiche :

L'element d'indice 0 du tableau est 1

L'element d'indice 1 du tableau est 5

L'element d'indice 2 du tableau est 2

Création de tableau

On va illustrer les méthodes en créant le tableau $[1, 4, 9, 16, \dots, n^2]$.

- Créer un tableau de longueur n et modifier un à un les éléments

```
n = ...  
t = n*[0]  
for i in range(n) :  
    t[i] = (i+1)**2
```

- Créer un tableau vide et insérer un à un les éléments

```
n = ...  
t = []  
for i in range(1,n+1) :  
    t.append(i**2)
```

- Utiliser une *list comprehension* : $[expr \text{ for } i \text{ in range}(\dots)]$ crée le tableau des valeurs successives de $expr$ lorsque i parcourt le range

```
n = ...  
t = [i**2 for i in range(1,n+1)]
```

Le pb de la duplication

```
>>> x = [4,2,5]
>>> y = x
>>> y += [3,9]
>>> x
>>> [4,2,5,3,9]
```

Explications :

- si x a pour valeur un tableau, x contient seulement l'adresse de l'emplacement mémoire du tableau (référence).
- $y = x$ implique la copie de l'adresse uniquement, donc les deux variables référencent le même tableau
- conséquence : une modification de x ou de y modifie les deux variables

Alternative : dupliquer ou cloner le tableau

2 types d'égalités

- égalité structurelle : les éléments sont égaux 2 à 2, testable avec "=="
- égalité physique : référencé au même endroit, une modification de l'un modifie l'autre, testable avec "is"

Rq : égalité physique \Rightarrow égalité structurelle

Solutions pour éviter l'égalité physique

- `y = list(x)`
- `y = x[:]`

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 **Données structurées**
 - Tableaux
 - **Dictionnaires**
 - Autres types de données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité

Problème : on souhaite enregistrer des informations (par exemple l'âge) sur des personnes (représentées par leur nom).

→ il faut un type de données structuré → on peut par exemple utiliser 2 tableaux : un pour les noms et un pour les âges avec une correspondance entre indice.

Mais : l'accès et la modification des informations nécessite une recherche coûteuse en temps.

Solution : utiliser des dictionnaires !

Définition 3.4

Un **dictionnaire** (*python*) est une table d'association clé-valeur.
Les clés sont toutes d'un même type et doivent être comparables.

Exemple

```
>>> d = {"Julien" : 18, "Virginie" : 19, "Matéo" : 18}
>>> d["Virginie"]
19
>>> d["Chloé"] = 21
>>> d
{"Julien" : 18, "Virginie" : 19, "Matéo" : 18, "Chloé" : 21}
```

Fonctions et méthodes sur les dictionnaires

taille d'un dictionnaire : `len(d)`

accès à la valeur d'une clé : `d[c]`

présence d'une clé : `c in d`

ajout d'une association : `d[c] = v`

parcourir un dictionnaire

```
for c in d :  
    print(d[c])
```

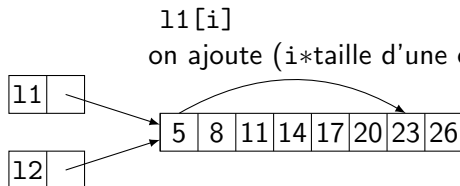
faire une copie de d1 : **Attention !**

~~d2 = d1~~ (sinon les dictionnaires sont dépendants comme les tableaux)

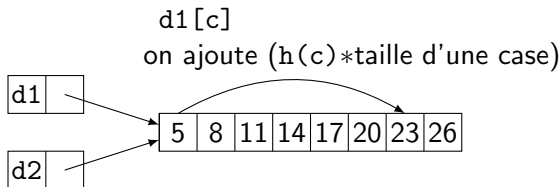
solution : `d2 = copy(d1)`

Fonctionnement des tableaux et dictionnaires

- Modèle mémoire des tableaux : cases mémoires consécutives



- Modèle mémoire des dictionnaires : **table de hachage**
→ un dictionnaire est associé à une **fonction de hachage** h .



Quand utiliser un dictionnaire ?

Un tableau peut être vu comme un dictionnaire dont les clés sont $\llbracket 0, n \rrbracket$

- si besoin de clés de types non entier
 - string
 - float
 - tous les types immuables (Contre-exemple : pas les listes)
- si besoin d'un ensemble de clés entières :
 - négatives
 - non-consécutives

Remarque : on peut implémenter un dictionnaire en utilisant un tableau de couples (clé,valeur), mais les opérations seront moins efficaces.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 **Données structurées**
 - Tableaux
 - Dictionnaires
 - **Autres types de données structurées**
- 4 Preuves sur les algorithmes
- 5 Récursivité

Tuples

Différences tuples - tableau

- on ne peut pas modifier, supprimer ou insérer un élément d'un tuple
- délimité par des parenthèses, pas par des crochets

```
>>> t = (7,2,9)
>>> len(t)
3
>>> class(t)
<class 'tuple'>
>>> t[1]
2
>>> u = (3,)          # tuple de longueur 1
```

Remarques :

- on peut concaténer les tuples
- couple : tuple de longueur 2
- souvent utiles pour renvoyer plusieurs valeurs.

Chaînes de caractères

Chaînes : similaires aux tuples - MAIS - chaque élément est une chaîne de longueur 1

- **Conséquences** : pas de modification, suppression ou insertion de lettre
- " " ou ' ' : pour pouvoir considérer l'un de ces symboles comme un symbole normal. ("l'espoir", mais pas 'l'espoir')

```
>>> s = 'le lac'
```

```
>>> s[1]
```

```
'e'
```

```
>>> for c in s :
```

```
...     print(c)
```

```
l
```

```
e
```

```
l
```

```
a
```

```
c
```

```
>>> len(s)
```

```
6
```

Symboles particuliers pour les chaînes en python

Codage	Interprétation
\\	\
\/	'
\"	"
\n	saut de ligne
\t	tabulation horizontale
\r	retour chariot

Exemple :

```
>>> s = 'Je demandais au Python :\ n\ t- Le lion ou l\
'antilope ?'
```

```
>>> print(s)
```

Je demandais au Python :

- Le lion ou l'antilope ?

Fonctions et méthodes sur les chaînes

- `join` : concatène les chaînes contenues dans une liste en intercalant un séparateur donné

```
>>> ch = ','.join(['Nifnif','Nafnaf','Noufnouf'])
>>> ch
'Nifnif,Nafnaf,Noufnouf'
```

- `split` : sépare une chaîne en sous-chaînes séparées par un séparateur donné

```
>>> ch.split(',')
['Nifnif','Nafnaf','Noufnouf']
```

- `list` : transforme la chaîne en liste de chaînes de longueur 1

```
>>> s = 'gorille'
>>> l = list(s)
>>> l
['g','o','r','i','l','l','e']
>>> l[0] = 'G'
>>> ''.join(l)
'Gorille'
```

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes**
 - Preuves de terminaison
 - Preuves de correction
- 5 Récursivité

Quelle est la différence entre ces deux programmes ?

algo1(n : entier)

```
1 tant que  $n \neq 0$   
   faire  
2   |   afficher ( $n$ )  
3   |    $n \leftarrow n-1$   
4 fin
```

algo2(n : entier)

```
1 tant que  $n > 0$   
   faire  
2   |   afficher ( $n$ )  
3   |    $n \leftarrow n-1$   
4 fin
```

De nombreux programmes ont des applications critiques :

- avionique
- médical
- véhicules autonomes
- sécurité des centrales nucléaires

L'erreur est humaine donc...

- on teste scrupuleusement les fonctions
... mais on ne peut pas être exhaustif
- on a besoin de prouver la correction des programmes critiques !
 - à la main (au programme)
 - ou automatiquement (hors programme)

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes**
 - Preuves de terminaison
 - Preuves de correction
- 5 Récursivité

Preuves de terminaison : les cas simples

Fonctions simples, contenant :

- instructions élémentaires
- instructions conditionnelles
- pas de boucle
- appels à des sous-fonctions sans boucle (sans récursivité)

⇒ pas de problème de terminaison.

Les boucles for

- avec range
 - on connaît le nombre d'itérations
- parcours d'une liste ou d'un dictionnaire
 - on connaît le nombre d'itérations
 - sauf si on modifie la structure dans la boucle (→ à éviter!!)

→ les preuves se concentrent sur les boucles while

Terminaison d'une boucle `while` : principe

Concept mathématique :

- Toute suite majorée d'entiers strictement croissante est finie.
 - Toute suite minorée d'entiers strictement décroissante est finie.
- On dit que \mathbb{N} est bien fondé.*

Principe des preuves de terminaison : **le variant de boucle**

- on exhibe une suite valeurs prises dans la boucle que l'on sait finie
- cette suite est appelée *variant de boucle*

Terminaison d'une boucle `while` : exemple 1

`premiere_occ(x : entier, t : entier[])`

Données : i, n : entiers

```
1  $n \leftarrow \text{taille}(t)$ 
2  $i \leftarrow 0$ 
3 tant que  $i < n$  et  $t[i] \neq x$  faire
4   |    $i \leftarrow i + 1$ 
5 fin
6 retourner  $(i < n)$ 
```

Preuve de terminaison :

- La suite des valeurs prises par i est :
 - entière
 - strictement croissante (ligne 4),
 - majorée ($i < n$ ligne 3).
- Cette suite est donc finie.
- Il y a donc un nombre fini d'itérations : **L'algorithme termine !**



Terminaison d'une boucle while : exemple 2

derniere_occ(x : entier, t : entier[])

Données : i, n : entiers

```
1  $n \leftarrow \text{taille}(t)$ 
2  $i \leftarrow n-1$ 
3 tant que  $i \geq 0$  et  $t[i] \neq x$  faire
4   |  $i \leftarrow i-1$ 
5 fin
6 retourner ( $i \geq 0$ )
```

Preuve de terminaison :





Terminaison d'une boucle while : exemple 3

euclide(a : entier, b : entier)

Données : c : entier

1 **tant que** $b > 0$ **faire**

2 $c \leftarrow b$

3 $b \leftarrow a \% b$

4 $a \leftarrow c$

5 **fin**

6 **retourner** a

Preuve de terminaison :



Terminaison d'une boucle `while` : généralisation

Variants de boucle composés

Les variants de boucles sont parfois composés de plusieurs variables :

- somme de plusieurs valeurs
- ordre lexicographique sur plusieurs valeurs, les listes ou les `string`
→ $(1,3,5) < (1,4,3) < (2,0,0) < (2,0,2) < (2,1,0)$

Terminaison d'une boucle while : exemple 4

```
lexico()


---


Données : t : entier[]
1 t ← [0,0]
2 tant que t[0] < 10 faire
3   | si t[1] < 9 alors
4   |   | t[1] ← t[1]+1
5   | sinon
6   |   | t ← [t[0]+1,0]
7   | fin
8   | afficher (t)
9 fin
```

Preuve de terminaison :

- La suite des couples des valeurs de t[0] et t[1] est :
 - une suite de couples d'entiers
 - strictement croissante pour l'ordre lexicographique,
 - majorée (t < [10,0] ligne 2).
- Cette suite est donc finie.
- Il y a donc un nombre fini d'itérations. **L'algorithme termine !**

Pas si facile en général : le problème de l'arrêt

Problème de l'arrêt

Existe-il un programme qui a la spécification suivante :

Entrée : un programme P et une entrée X du programme

Sortie : oui si P termine sur l'entrée X et non sinon

- S'il existe un programme A qui résout ce problème

→ on peut écrire un programme B :

```
def B(P) :  
    si A(P,P) alors : boucle infinie  
    sinon : oui
```

- Que se passe-t-il lorsqu'on appelle $B(B)$?
 - Si $A(B, B)$ est vrai ($\Leftrightarrow B(B)$ termine), alors $B(B)$ rentre dans une boucle infinie. → **Absurde**
 - Si $A(B, B)$ est faux ($\Leftrightarrow B(B)$ ne termine pas), alors $B(B)$ termine. → **Absurde**

→ Il n'existe pas un tel programme A , on dit que ce problème est **indécidable**.

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes**
 - Preuves de terminaison
 - **Preuves de correction**
- 5 Récursivité

Preuves de correction : approche générale

- **Invariants de boucle** : on prouve des propriétés qui restent vraies à chaque itération de boucle
- **Terminaison** : on prouve qu'on finit par sortir de la boucle
- **Preuve de correction** : combinaison des deux pour prouver que l'algorithme fait ce qu'on attend.

Principe d'une preuve de correction :

$$\left. \begin{array}{l} \text{invariant(s)} \\ \text{condition de fin de boucle} \end{array} \right\} \Rightarrow \text{preuve du résultat}$$

Preuves de correction : exemple 1

appartient(x : entier, l : entier[])

Données : i : entier

```
1 pour  $i$  de 0 à  $\text{taille}(l)-1$  faire
2   |   si  $l[i] = x$  alors
3     |   retourner Vrai
4   |   fin
5 fin
6 retourner Faux
```

Preuve :

La preuve de terminaison est immédiate car il y a un **pour**.

On *admet* l'invariant de boucle suivant :

(I_1) si l'itération d'indice i a lieu, $l[0:i]$ ne contient pas x

Preuves de correction : exemple 1

appartient(x : entier, l : entier[])

Données : i : entier

```
1 pour  $i$  de 0 à  $\text{taille}(l)-1$  faire
2   |   si  $l[i] = x$  alors
3   |   |   retourner Vrai
4   |   fin
5 fin
6 retourner Faux
```

À la sortie de la boucle :

- ① soit $l[i] = x$
- ② soit $i = \text{taille}(l)$ (fin de la boucle for)
 - **cas 1** : le résultat renvoyé est Vrai ✓
 - **cas 2** : (I_1) avec $i = \text{taille}(l)$ implique que $l[0:\text{taille}(l)]$ (c-à-d tout l) ne contient pas x ; et le résultat renvoyé est Faux ✓

Conclusion : l'appel `appartient(x,l)` renvoie le résultat attendu.

maximum(t : entier[])

Données : m, i : entiers

```
1  $m \leftarrow t[0]$ 
2 pour  $i$  de 1 à  $\text{taille}(t)-1$  faire
3   |   si  $t[i] > m$  alors
4   |   |    $m \leftarrow t[i]$ 
5   |   fin
6 fin
7 retourner  $m$ 
```

Preuve :

Notons m_i la valeur de m avant l'itération d'indice i .
On *admet* l'invariant de boucle suivant :

$$(I_2) \quad m_i = \max(t[0 : i])$$

Preuves de correction : exemple 2

maximum(t : entier[])

Données : m, i : entiers

```
1  $m \leftarrow t[0]$ 
2 pour  $i$  de 1 à  $\text{taille}(t)-1$  faire
3   |   si  $t[i] > m$  alors
4   |   |    $m \leftarrow t[i]$ 
5   |   fin
6 fin
7 retourner  $m$ 
```

$$(I_2) \quad m_i = \max(t[0 : i])$$

Conclusion :

Preuves d'invariants : exemple 1

appartient(x : entier, l : entier[])

Données : i : entier

```
1 pour  $i$  de 0 à  $\text{taille}(l)-1$  faire
2   | si  $l[i] = x$  alors
3   |   | retourner Vrai
4   | fin
5 fin
6 retourner Faux
```

(I_1) si l'itération d'indice i a lieu, $l[0:i]$ ne contient pas x

Preuve :

(I) pour $i = 0$: $l[0:0]$ est vide, donc ne contient pas x

(H) Soit i_0 tq l'invariant est vrai. Supp. qu'une nouvelle itération a lieu.

Par hypothèse de récurrence, x n'appartient pas à $l[0:i_0]$.

De plus, si l'itération a lieu avec $i = i_0 + 1$, cela implique que le test $l[i] == x$ était faux pour $i = i_0$.

Ainsi, x n'appartient pas à $l[0:i_0 + 1]$.

(C) L'invariant est bien vrai pour toute itération de la boucle.

Preuves d'invariants : exemple 2

maximum(t : entier[])

Données : m, i : entiers

```
1  $m \leftarrow t[0]$ 
2 pour  $i$  de 1 à  $\text{taille}(t)-1$  faire
3   |   si  $t[i] > m$  alors
4   |   |    $m \leftarrow t[i]$ 
5   |   fin
6 fin
7 retourner  $m$ 
```

Notons m_i la valeur de m avant l'itération d'indice i .

$$(I_2) \quad m_i = \max(t[0 : i])$$

Preuve :

(I)

(H)

(C)

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité**
 - Fonctions récursives
 - Terminaison
 - Correction

Exemple introductif : tours de Hanoi



Objectif

Déplacer les disques de la barre de gauche vers la barre de droite.

Règles

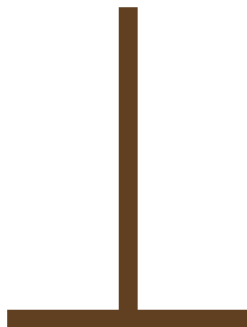
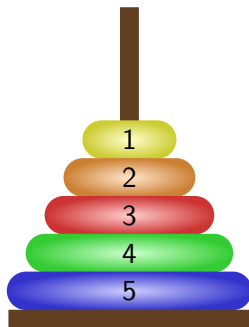
- On ne peut prendre qu'un disque à la fois.
- On ne peut placer un disque que sur un disque plus grand.

Exemple introductif : tours de Hanoi





Généralisation ?



Formalisation

On a exprimé la solution d'un problème à *partir* de solutions de *sous-problèmes*.

La démarche de décrire la solution d'un problème comme combinaison de solutions de problèmes identiques de taille plus petite s'appelle **la récursivité**.

Récursivité

Donner une solution récursive à un problème consiste à :

- Exprimer la solution du problème à partir de sous-problèmes :

$$\text{hanoi}(n+1,1,3) = \text{hanoi}(n,1,2) + m(1,3) + \text{hanoi}(n,2,3)$$

$$\text{hanoi}(n+1,1,2) = \dots$$

$$\text{hanoi}(n+1,2,3) = \dots$$

- Donner la solution dans *les cas de bases* :

$$\text{hanoi}(1,1,3) = m(1,3)$$

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 **Réversibilité**
 - Fonctions récursives
 - Terminaison
 - Correction

Définition 5.1

Une **fonction récursive** est une fonction dont le calcul appelle la fonction elle-même.

Factorielle

$$\begin{cases} 0! = 1 \\ 1! = 1 \\ n! = n \times (n-1)! \quad \text{pour } n \geq 2 \end{cases}$$

Attention !

Il est indispensable d'avoir des cas de base (« $0! = 1$ » et « $1! = 1$ » dans l'exemple précédent).

Sans ces cas de base, la fonction s'appellera indéfiniment :

Avec la seule définition « $n! = n \times (n-1)!$ », on aurait :

$$3! = 3 \times 2 \times \cdots \times (-5) \times (-6) \times \cdots$$

- Définition récursive de l'addition (avec $a, b \in \mathbb{N}^2$) :

$$ajoute(a, b) = \left\{ \begin{array}{l} \end{array} \right.$$

Algorithme associé :

- Une nouvelle façon d'itérer (similaire au while)

ajoute(a,b : entiers)

```
1 tant que  $b > 0$  faire
2   |    $a \leftarrow a+1$ 
3   |    $b \leftarrow b-1$ 
4 fin
5 retourner a
```

ajoute(a,b : entiers)

```
1 si  $b > 0$  alors
2   |   retourner ajoute(a+1,b-1)
3 sinon
4   |   retourner a
5 fin
```

Les mêmes questions se posent

- terminaison ?
- correction ?

- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité**
 - Fonctions récursives
 - **Terminaison**
 - Correction

Les preuves de terminaison fonctionnent comme celles des boucles `while`.

```
ajoute(a,b : entiers)
1 si  $b > 0$  alors
2   |   retourner ajoute(a+1,b-1)
3 sinon
4   |   retourner a
5 fin
```



- 1 Concepts de bases
- 2 Structuration d'un programme
- 3 Données structurées
- 4 Preuves sur les algorithmes
- 5 Récursivité**
 - Fonctions récursives
 - Terminaison
 - Correction

Les preuves de correction se font par récurrence.

```
ajoute(a,b : entiers)
1 si b > 0 alors
2   | retourner ajoute(a+1,b-1)
3 sinon
4   | retourner a
5 fin
```

(I)

(H)

(C)