# Cours Algorithmique

Samba Ndojh NDIAYE

**Laboratoire d'InfoRmatique en Image et Systèmes d'information**
LIRIS UMR 5205 CNRS/Université Claude Bernard Lyon 1
samba-ndojh.ndiaye@univ-lyon1.fr

# Évaluation

## Notes

- 1 DS de promotion
- Contrôles en continu (au moins 2)
- Note finale : 2/3 DS + 1/3 CC ou 1/2 DS + 1/2 CC

# Supports

## Supports

- Cours : https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/Cours_algo/cours_algo.pdf
- Slides : `https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/Cours_algo/slides_algo.pdf`
- Recueil d'exercices avec auto-évaluation : `https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/exercices/`
- Python Tutor (visualisation de traces d'exécution) : `https://pythontutor.com/`

# Plan

# Plan

# Algorithme

## Algorithmique

L'algorithmique peut se définir comme "une méthode de résolution d'un problème sous la forme d'une suite d'opérations élémentaires obéissant à un enchaînement déterminé".

## Exemple

On considère une pile de dossiers classés par ordre alphabétique sur le nom. Donner un algorithme qui détermine si une personne de nom X a bien un dossier à son nom dans la pile.

# Spécification formelle

- les données du problème (les entrées) ;
- les résultats (les sorties) ;
- les éventuelles préconditions sur les données (des informations sur la nature des données) ;
- les postconditions (une relation entre entrées et sorties).

# Spécification formelle

**Algorithme 1** : Procédure : Recherche linéaire

**Entrées** : Une pile de dossiers **P**
Un nom **X**
**Sorties** : Une réponse **R**
**Précondition :** Les dossiers de **P** sont empilés par ordre alphabétique sur le nom, le dossier comportant le plus petit nom se trouvant au sommet de la pile.
**Postcondition :** Si **P** contient un dossier au nom de **X** alors **R** doit être égal à **oui** sinon **R** doit être égal à **non**.

# Recherche linéaire

## **Algorithme 2** : Procédure : Recherche linéaire

**Entrées** : Une pile de dossiers **P**
Un nom **X**
**Sorties** : Une réponse **R**
**Précondition :** Les dossiers de **P** sont empilés par ordre alphabétique sur le nom, le dossier comportant le plus petit nom se trouvant au sommet de la pile.
**Postcondition :** Si **P** contient un dossier au nom de **X** alors **R** doit être égal à **oui** sinon **R** doit être égal à **non**.

```
début
    tant que la pile de dossiers P n'est pas vide et le nom du dossier au sommet de P
    est inférieur à X faire
        Prendre le dossier au sommet de P
        Le poser à côté de P
    fintq
    si P est vide ou le nom du dossier au sommet de P est différent de X alors
        R ← non
    sinon
        R ← oui
    finsi
fin
```

# Propriétés

## Opérations élémentaires

La nature des opérations élémentaires utilisées pour résoudre le problème dépend des capacités de la personne (ou machine) qui va exécuter l'algorithme.

## Terminaison

Cet algorithme s'exécute en un temps fini : combien d'opérations ? Peut-on faire mieux ?

# Recherche dichotomique

## **Algorithme 3** : Procédure : Recherche dichotomique

**Entrées** : Une pile de dossiers **P**
Un nom **X**
**Sorties** : Une réponse **R**
**Précondition :** Les dossiers de **P** sont empilés par ordre alphabétique sur le nom, le dossier comportant le plus petit nom se trouvant au sommet de la pile.
**Postcondition :** Si **P** contient un dossier au nom de **X** alors **R** doit être égal à **oui** sinon **R** doit être égal à **non**.

```
début
    tant que la pile de dossiers P n'est pas vide et le nom du dossier au sommet de P est inférieur à X faire
        Couper la pile de dossier en 2 parties;
        si le nom du dossier au sommet de P_inf est X alors
            Ne garder que le premier dossier de P_inf
        sinon si le nom du dossier au sommet de P_inf est supérieur à X alors
            Enlever de P tous les dossiers de P_inf
        sinon
            Enlever de P le premier dossier de P_inf ;
            Enlever de P tous les dossiers de P_sup
        finsi
    fintq
    si P est vide alors
        R ← non
    sinon
        R ← oui
    finsi
fin
```

# Complexité

## Complexité en temps

La complexité en temps d'un algorithme est une évaluation en ordre de grandeur du nombre d'opérations élémentaires qu'il peut réaliser (dans le pire des cas).
Elle est déterminée en fonction des entrées.

## Exemples

Calcul de la complexité des Recherches linéaire et dichotomique.

# Plan

# Du problème à la spécification formelle

## Spécification formelle

À partir de la description informelle, ambiguë et incomplète du problème, il faut élaborer une spécification formelle qui précise :

- les paramètres en entrée
- les paramètres en sortie
- les préconditions
- les postconditions
- les contraintes à respecter pour la résolution du problème (notamment les contraintes de ressources).

# De la spécification formelle à l'algorithme

## Algorithme

À partir de la spécification formelle, on élabore un algorithme qui spécifie l'enchaînement d'opérations élémentaires à effectuer pour résoudre le problème.

## Démarche descendante

Un algorithme est généralement élaboré selon une démarche descendante, qui consiste à décomposer le problème en sous-problèmes, chaque sous-problème devant être de nouveau clairement spécifié puis résolu.

# Vérification de l'algorithme

## Vérification

Il s'agit de vérifier que l'algorithme répond effectivement au problème spécifié, et donc qu'il est :

- **correct**, c'est-à-dire que les valeurs des paramètres de sortie calculées par l'algorithme sont effectivement celles que l'on souhaitait calculer ;
- **complet**, c'est-à-dire que les bonnes valeurs des paramètres de sortie sont trouvées pour toutes les valeurs possibles des paramètres en entrée ;
- **fini**, c'est-à-dire que le nombre d'opérations élémentaires à effectuer pour résoudre le problème est fini.

# Vérification de l'algorithme

## Vérification théorique

La vérification peut être effectuée de façon théorique par une preuve mathématique.

## Vérification expérimentale

La vérification est généralement faite par des tests sur un certain nombre de données appelées jeux d'essai.

# De l'algorithme au programme

## Codage

L'algorithme est codé dans un langage de programmation en s'adaptant aux opérations élémentaires définies par celui-ci.

# Du programme au code exécutable

## Compilation et interprétation

Le code écrit dans un langage de programmation (code source) est traduit par un compilateur ou un interpréteur en un code en langage machine (code objet) exécutable par l'ordinateur.

# Plan

# Langage algorithmique

## Python

- L'écriture d'un algorithmique est généralement faite dans un langage semi-formel, pseudo-code, permettant d'être moins ambiguë que le langage naturel.
- La richesse de ce langage est définie par l'ensemble des opérations qui peuvent être jugées comme étant élémentaires.
- Nous allons utiliser directement un langage de programmation, le langage Python qui nous donnera les capacités initiales de l'ordinateur, au-dessus desquelles nous allons construire nos algorithmes.

# Spécification formelle en python

## Spécification

La spécification a pour but de faciliter l'utilisation du code et son éventuelle modification.

```
def Recherche_linéaire(P) :
    """
     :entree P : une pile de dossiers
     :pré-cond : Les dossiers de P sont empilés par ordre alphabétique sur le nom,
                 le dossier comportant le plus petit nom se trouvant au sommet de la pile.
     :entree X : un nom
     :sortie R : Une réponse
     :post-cond : Si P contient un dossier au nom de X alors R doit être
                  égal à oui, sinon R doit être égal à non.
    """
```

# Plan

# Plan

# Nombres entiers : int

## Exemples

0, 42, +123, -987654

- >>> 10+3 # addition
  13
- >>> 10-3 # soustraction
  7
- >>> 10*3 # multiplication
  30
- >>> 10//3 # division entière
  3
- >>> 10%3 # modulo
  1
- >>> 10**3 # puissance
  1000

# Nombres entiers : int

## Remarques

- règles de priorité habituelles entre opérateurs
- priorité élevée des parenthèses
- l'opérateur de division entière // est différent de l'opérateur de division réel /.
- calculs sur des entiers arbitrairement grands.

# Nombres flottants : float

## Exemples

0.0, 4.2, +12.3, -987.654, 1e2, -3.4e+5, 6.7e-8.

- >>> 2.5 + 1.5 # addition
  4.0
- >>> 2.5 - 1.5 # soustraction
  1.0
- >>> 2.5 * 1.5 # multiplication
  3.75
- >>> 2.5 / 1.5 # division réelle
  1.6666666666666667
- >>> 2.5 ** 1.5 # puissance
  3.952847075210474

# Nombres flottants : float

**Remarque**

Si ces opérations combinent un entier et un flottant, le résultat sera un flottant.

# Booléens : bool

## Booléens

- Il n'existe que deux valeurs booléennes : True et False.
- Il n'existe que trois opérateurs sur les booléens : and, or et not.

- >>> False and True
  False
- >>> False or True
  True
- >>> not False
  True

# Opérateurs de comparaison

- >>> 1+1 == 2 # égalité
  True
- >>> 1+1 != 2 # différence
  False
- >>> 3.3 < 10/3 # inférieur strict
  True
- >>> 3.3 > 10/3 # supérieur strict
  False
- >>> 1+1 <= 2 # inférieur ou égal
  True
- >>> 1+1 >= 2 # supérieur ou égal
  True
- >>> (1+1 == 2) and (3.3 > 10/3)
  False

# Chaînes de caractères : str

## Exemples

'hello', "bonjour le monde".

- >>> len("le monde") # longueur
  8
- >>> "le" + "monde" # concaténation
  'lemonde'
- >>> "le monde"[0] # premier caractère
  'l'
- >>> "le monde"[ :3] # sous-chaîne des 3 premiers caractères
  'le '
- >>> "le monde"[3 :] # sous-chaîne à partir du 4ème caractère
  'monde'

# Chaînes de caractères : str

## Remarques

- Pas de type caractère distinct du type chaîne : un caractère est une chaîne de longueur 1
- Les chaînes de caractères sont indicées à partir de 0
- Les chaînes de caractères peuvent être comparées avec les opérateurs de comparaison : ordre lexicographique
- Les chiffres sont des caractères valides : éviter les confusions entre les chaînes de caractères composées de chiffres avec les nombres correspondants

# Chaînes de caractères : str

- >>> "le monde" == "le " + "monde"
  True
- >>> "bonjour" < "bonsoir"
  True
- >>> "Z" < "a"
  True
- >>> "é" >= "z"
  True
- >>> "123" == 123
  False
- >>> "1" + "1"
  '11'
- >>> "10" < "2"
  True

# Plan

# Variables

## Variables

- Une variable est un emplacement mémoire, muni d'un nom et contenant une valeur (qui peut changer au fil du temps).
- L'opération d'affectation consiste à fixer ou changer la valeur d'une variable.
- L'opérateur d'affectation est =
- À gauche de l'opérateur d'affectation, on indique le nom de la variable à affecter.
- À droite de l'opérateur d'affectation, on donne une expression dont la valeur est affectée à la variable

# Variables : affectation

- >>> message = "bonjour le monde"
- >>> message
  'bonjour le monde'
- >>> a = 42
- >>> a+1
  43
- >>> a = a+1
- >>> a
  43

# Variables

## Remarques

- les variables en python ne sont ni déclarées, ni typées
- la première affectation de la variable fait office de déclaration
- il est possible pour une variable de contenir, à deux moments différents, des valeurs de types différents : à éviter.

# Plan

# Plan

# Fonction

```
def encadre_réel(x : float) -> (int, int) :
    """
     :entrée x : float
     :pré-cond : x ≥ 0
     :sortie inf : int
     :sortie sup : int
     :post-cond : inf ≤ x < sup
                  |sup − inf| ≤ 1
    """
    # (...) enchaînement d'instructions
    return inf, sup
```

# Fonction

## Procédure

Une procédure est une fonction qui n'a pas d'instruction return, ou dont l'instruction return n'est suivie d'aucune expression.

# Variables d'une fonction

## Variables d'une fonction

Il peut exister trois catégories de variables dans une fonction.

- les variables correspondants aux paramètres d'entrée ;
- les variables correspondants aux paramètres de sortie ;
- les variables locales.

# Variables d'une fonction

## variables correspondants aux entrées

Elles ont déjà une valeur au début de la fonction. Elles n'ont donc pas besoin d'être affectées, et on évitera en général de changer leur valeur.

## variables correspondants aux sorties

Elles doivent être affectées dans la fonction dont c'est le rôle de déterminer leurs valeurs. Ces valeurs sont transmises à l'appelant par l'instruction return à la fin de la fonction.

## variables locales

Ce sont toutes les autres variables qui peuvent être nécessaires au calcul des sorties. Elles n'ont pas de valeur initialement, et leur valeur est " oubliée " à la fin de la fonction.

# Plan

# Séquence

## Enchaînement en séquence

Les instructions sont écrites l'une après l'autre, séparées par un saut de ligne. Elles sont toutes exécutées, dans l'ordre ou elles sont écrites.

---

**def** puissances($x$ : float) -> (float, float, float) :

"""

:entrée $x$ : float
:pré-cond : $\emptyset$
:sortie $p2$ : float
:sortie $p3$ : float
:sortie $p4$ : float
:post-cond : $p2 = x^2$, $p3 = x^3$, $p4 = x^4$

"""

$p2 = x * x$
$p3 = p2 * x$
$p4 = p3 * x$
return $p2$, $p3$, $p4$

---

# Plan

# Condition

## Enchaînement conditionnel

Des instructions différentes sont exécutées selon qu'une condition est remplie ou non.

```
def fonction(...) :
    """
    ...
    """
    if condition :
            # suite d'instructions à exécuter si la condition est vérifiée
    else :
            # suite d'instructions à exécuter si la condition n'est pas vérifiée
```

# Condition

## Remarques

- la condition est sous la forme d'une expression booléenne
- La clause else et les instructions associées sont facultatives
- Les suites d'instructions associées aux clauses if et else ne sont pas limitées à des enchaînements séquentiels
- La première instruction précédée du même nombre d'espaces (ou moins) que la première ligne (la ligne du if) sera reconnue comme ne faisant pas partie de l'enchaînement conditionnel

# Condition : exemple

```
def valabs_et_signe(x : float) -> (int, float) :
    """
     :entrée x : float
     :pré-cond : ∅
     :sortie signe : int
     :sortie valabs : float
     :post-cond : signe = 1 si x ≥ 0, −1 sinon
     :post-cond : valabs = |x|
    """
    if x >= 0  :
                signe = 1
                valabs = x
    else :
                signe = −1
                valabs = −x
    return signe, valabs
```

# Plan

# Répétition : while

## Répétition

Python supporte les enchaînements répétitifs, également appelés "enchaînements itératifs" ou "boucles".

## while

La boucle while permet de répéter des instructions tant qu'une condition est remplie.

```
def fonction(...) :
    """
    ...
    """
    while condition :
        # suite d'instructions à exécuter tant que la condition est vérifiée
```

# while : exemple

```
def nb_chiffres(n : int) -> int :
    """
     :entrée n : int
     :pré-cond : n ≥ 0
     :sortie c : int
     :post-cond : n s'écrit en base 10 avec c chiffres
    """
    c = 1
    while n >= 10  :
                    c = c + 1
                    n = n//10
    return c
```

# while : exemple 2

```
def factorielle(n : int) -> int :
    """
     :entrée n : int
     :pré-cond : n ≥ 0
     :sortie f : int
     :post-cond : f = n! = 1 * 2 * 3 * ... * (n − 1) * n
    """
    f = 1
    i = 1
    while i < n  :
                  i = i + 1
                  f = f * i
    return f
```

# exécution pour n = 4

| nb de passages dans la boucle | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| valeur de i | 1 | 2 | 3 | 4 |
| valeur de f | 1 | 2 | 6 | 24 |

# Boucle while et terminaison

## while

La terminaison de cet algorithme est assurée par le fait que la valeur de i augmente de 1 à chaque passage dans la boucle while et que l'on s'arrête quand elle devient supérieure ou égale à n.

# Complexité et correction

## Complexité

La complexité de factorielle est linéaire.

## Correction

La correction de l'algorithme peut être démontrée à l'aide d'une propriété invariante.
À chaque passage dans la boucle while, la propriété $f = i!$ est vérifiée.
En sortant de la boucle, $i = n$, et donc $f = i! = n!$.

# Conseils sur les enchaînements répétitifs

## Blocs à identifier

- Instructions d'initialisation
- Condition d'arrêt : c'est la condition qui doit être satisfaite pour arrêter de boucler.
- Instructions à répéter : ce sont les instructions qui sont exécutées à chaque passage dans la boucle. On distinguera généralement deux sous blocs dans ce bloc d'instructions : les instructions "de traitement" et les instructions "de passage".
- Les instructions "de traitement" sont celles à l'origine du fait que l'on écrit un enchaînement répétitif.
- Les instructions "de passage" sont celles qui permettent de modifier les variables sur lesquelles porte la condition d'arrêt (les variables qui permettent de contrôler le nombre de passages dans la boucle).

# Conseils sur les enchaînements répétitifs

## Questionnement

- Est-on certain que la condition d'arrêt est atteinte ?
- Combien de fois les instructions à répéter sont-elles exécutées ?
- Quelles sont les valeurs des variables lorsqu'on sort de la boucle ?

# Plan

# Appel de fonction

## Affectation

- la partie gauche comportera autant de variables que la fonction comporte de paramètres de sorties ;
- la partie droite est constituée du nom de la fonction, suivi par la liste des valeurs effectives des paramètres d'entrée (arguments), entre parenthèses et séparées par des virgules.
- >>> i, j, k = puissances(8)
- >>> k = factorielle(i)

# Appel de fonction

## Affectation

- la partie gauche comportera autant de variables que la fonction comporte de paramètres de sorties ;
- la partie droite est constituée du nom de la fonction, suivi par la liste des valeurs effectives des paramètres d'entrée (arguments), entre parenthèses et séparées par des virgules.
- les arguments d'un appel de fonctions sont des expressions (complexes)
- les variables recevant les valeurs des sorties ne sont pas tenues d'avoir le même nom que ces sorties

# Appel de fonction

## Affectation

- la partie gauche comportera autant de variables que la fonction comporte de paramètres de sorties ;
- la partie droite est constituée du nom de la fonction, suivi par la liste des valeurs effectives des paramètres d'entrée (arguments), entre parenthèses et séparées par des virgules.

## Expression

- L'appel d'une fonction avec une seule sortie peut être utilisé directement dans une expression
- >>> j = factorielle(0)+factorielle(1)+factorielle(2)

# Appel de fonction

## Affectation

- la partie gauche comportera autant de variables que la fonction comporte de paramètres de sorties ;
- la partie droite est constituée du nom de la fonction, suivi par la liste des valeurs effectives des paramètres d'entrée (arguments), entre parenthèses et séparées par des virgules.

## Expression

- L'appel d'une fonction avec une seule sortie peut être utilisé directement dans une expression

## Procédure

- L'appel d'une procédure se limitera au nom de la procédure suivi de la liste de ses arguments.
- >>> print("bonjour le monde")

# Portée des variables

## Affectation

- Les variables utilisées dans une fonction sont propres à cette fonction.
- Elles ne sont ni visibles, ni utilisables depuis d'autres fonctions.
- La portée d'une variable est limitée à la fonction qui la définit.

# Exemple

```
def total_chiffres_fact(n : int) -> int :
    """
     :entrée n : int
     :pré-cond : n > 0
     :sortie c : int
     :post-cond : c est le nombre total de chiffres nécessaires pour écrire
                  les factorielles de tous les entiers entre 1 et n
    """
    f = 1
    c = 0
    i = 1
    while i <= n :
                  f = f * i
                  c = c+nb_chiffres(f)
                  i = i + 1
    return c
```

# Exemple

```
def nb_chiffres(n : int) -> int :
    """
     :entrée n : int
     :pré-cond : n ≥ 0
     :sortie c : int
     :post-cond : n s'écrit en base 10 avec c chiffres
    """
    c = 1
    while n >= 10  :
                    c = c + 1
                    n = n//10
    return c
```

# Plan

# Plan

# Tableaux

## Définition

- Python préfère les listes
- module Numpy (http ://numpy.scipy.org/) à installer
- from numpy import *
- un tableau est une liste ordonnée de n valeurs du même type
- n est la taille du tableau
- les valeurs du tableau sont ses éléments
- chaque élément est repéré dans le tableau par son indice (un entier compris entre 0 et n-1)

# Tableaux

## Déclaration

- >>> from numpy import *
- >>> array([5,3,2,1,1]) # crée un tableau à partir de valeurs
  array([5, 3, 2, 1, 1])
- >>> zeros(3, float) # crée un tableau rempli de 0
  array([ 0., 0., 0.])
- >>> empty(4, int) # crée un tableau non initialisé
  array([0, 8826784, 31983376,0])

# Tableaux

## Opérations

- >>> a = array([5,3,2,1,1])
- >>> len(a) # longueur
  5
- >>> a[0] # premier élément
  5
- >>> a[ :3] # sous-tableau de 0 à 2
  array([5, 3, 2])
- >>> a[3 :] # sous-tableau de 3 à la fin
  array([1, 1])

# Tableaux

## Opérations

- >>> a = array([5,3,2,1,1])
- >>> a[1] = 9 # modification du 2ème élément du tableau
- >>> a
  array([5, 9, 2, 1, 1])

# Plan

# Tableaux et mutabilité

## Objets non mutables

Les opérations sur les entiers, flottants, booléens et chaînes produisent une nouvelle valeur à partir des opérandes. Chaque affectation d'une variable remplace sa valeur par une autre. Ainsi, la modification d'une variable ne peut pas avoir d'impact sur une autre.

- >>> a = 42
- >>> b = a # b prend la même valeur que a, c.à.d. 42
- >>> a = a+1 # a+1 produit la valeur 43, qui remplace 42 dans la variable a
- >>> a
  43
- >>> b # b, en revanche, contient toujours 42
  42

# Tableaux et mutabilité

## Mutabilité

Pour les tableaux, l'affectation d'un élément modifie l'état du tableau sans remplacer ce dernier par un autre tableau (les tableaux sont des objets mutables). Ainsi, si deux variables font référence au même tableau, une modification sur l'une des variables sera répercutée sur l'autre.

# Tableaux et mutabilité

## Mutabilité

- >>> a = array([5,3,2,1,1])
- >>> b = a # a et b font référence au MÊME tableau
- >>> b
  array([5, 3, 2, 1, 1])
- >>> a[1] = 9
- >>> a # l'état du tableau a été modifié...
  array([5, 9, 2, 1, 1])
- >>> b # ... ce qui se voit aussi sur b, puisqu'il s'agit du même tableau
  array([5, 9, 2, 1, 1])

# Tableaux et mutabilité

## Mutabilité

Les sous-tableaux produits par les opérations a[ :i] et a[i :] partagent également l'état du tableau qui a servi à les produire. Ce ne sont pas des copies partielles, mais des vues restreintes du tableau global. Ainsi, si on modifie un élément du sous-tableau, le tableau global est également modifié (et inversement)

# Tableaux et mutabilité

## Mutabilité

- >>> a = array([5,3,2,1,1])
- >>> b = a[3 :]
- >>> b
  array([1, 1])
- >>> b[1] = 7
- >>> b
  array([1, 7])
- >>> a # la modification de b est répercutée sur a
  array([5, 3, 2, 1, 7])

# Tableaux et mutabilité

## Mutabilité

- >>> a[3] = 8
- >>> a
  array([5, 3, 2, 8, 7])
- >>> b # la modification de a est répercutée sur b
  array([8, 7])

# Paramètres d'entrée-sortie

## entrée-sortie

Les objets mutables correspondent à des entrées-sorties et ils sont seuls à pouvoir correspondre aux entrées-sorties. L'état initial d'une entrée-sortie décrit tout ou une partie du problème et son état final tout ou une partie de la solution.

## exemple

:entrée/sortie $a$ : tableau de flottants
:pré-cond : $\emptyset$
:post-cond : pour tout $i$ entre $0$ et $len(a) - 1$, $a_s[i] == a_e[i]/2$

## entrée-sortie

Les entrées-sorties ne sont pas retournées : leurs modifications sont accessibles directement dans l'objet passé en paramètre.

# Plan

# Tableaux de chaînes de caractères

## Tableaux de chaînes

La taille maximale de leurs éléments doit être connue à la création du tableau.

- >>> a = array(["un", "deux"])
- >>> a[0] = "autre chose"
- >>> a[0]
  'autr'

# Tableaux de chaînes de caractères

## Tableaux de chaînes

La taille maximale de leurs éléments doit être connue à la création du tableau.

Les fonctions zeros et empty fixe par défaut cette taille à un caractère.

On peut fixer cette taille en utilisant le préfixe <U :

- a = zeros(10, "<U1000")

# Plan

# Plan

# La récursivité

## Définition

La récursivité peut se définir comme la résolution d'un problème à partir de versions plus simples de lui-même.

## Exemple : factorielle

1. définition de factorielle pour un cas élémentaire (règle de base) : $0\,! = 1$
2. définition de factorielle pour un cas non élémentaire, en fonction de la définition de factorielle pour un cas plus simple (règle récursive) : $n\,! = n * (n - 1)\,!$ pour $n \geq 1$

# La récursivité

## Exemple : fibonacci

1. première règle de base : fibo(0) $= 1$
2. deuxième règle de base : fibo(1) $= 1$
3. règle récursive : fibo(n) $=$ fibo(n - 1) $+$ fibo(n - 2) pour $n \geq 2$

# La récursivité

## Terminaison

La terminaison de ce genre de définition est garantie par le fait qu'un "critère" (en l'occurrence la valeur de n) est modifié d'un appel récursif à l'autre et converge (en un nombre d'étapes finies) vers un des cas de base.

# La récursivité

## Algorithme récursif

On procède toujours selon les deux étapes suivantes :

1. résolution du problème dans les cas élémentaires (les cas de base) ;
2. résolution du problème pour les cas non élémentaires en faisant appel à la résolution du problème pour des cas plus simples.

## Terminaison

Pour s'assurer de la terminaison de ce genre d'algorithme, il faut vérifier que d'un appel récursif à l'autre, les valeurs d'un ou plusieurs paramètres changent de telle sorte que l'on converge, en un nombre fini d'appels, vers un cas élémentaire.

# factorielle

```
def factorielle_rec(n : int) -> int :
    """
     :entrée n : int
     :pré-cond : n ≥ 0
     :sortie f : int
     :post-cond : f = n! = 1 * 2 * 3 * ... * (n − 1) * n
    """
    if n == 0  :
                f = 1
    else :
                f = n* factorielle_rec(n − 1)
    return f
```