M313
Algorithmique Avancée

Denis Pallez
Maitre de Conférences
http://denispallez.i3s.unice.fr

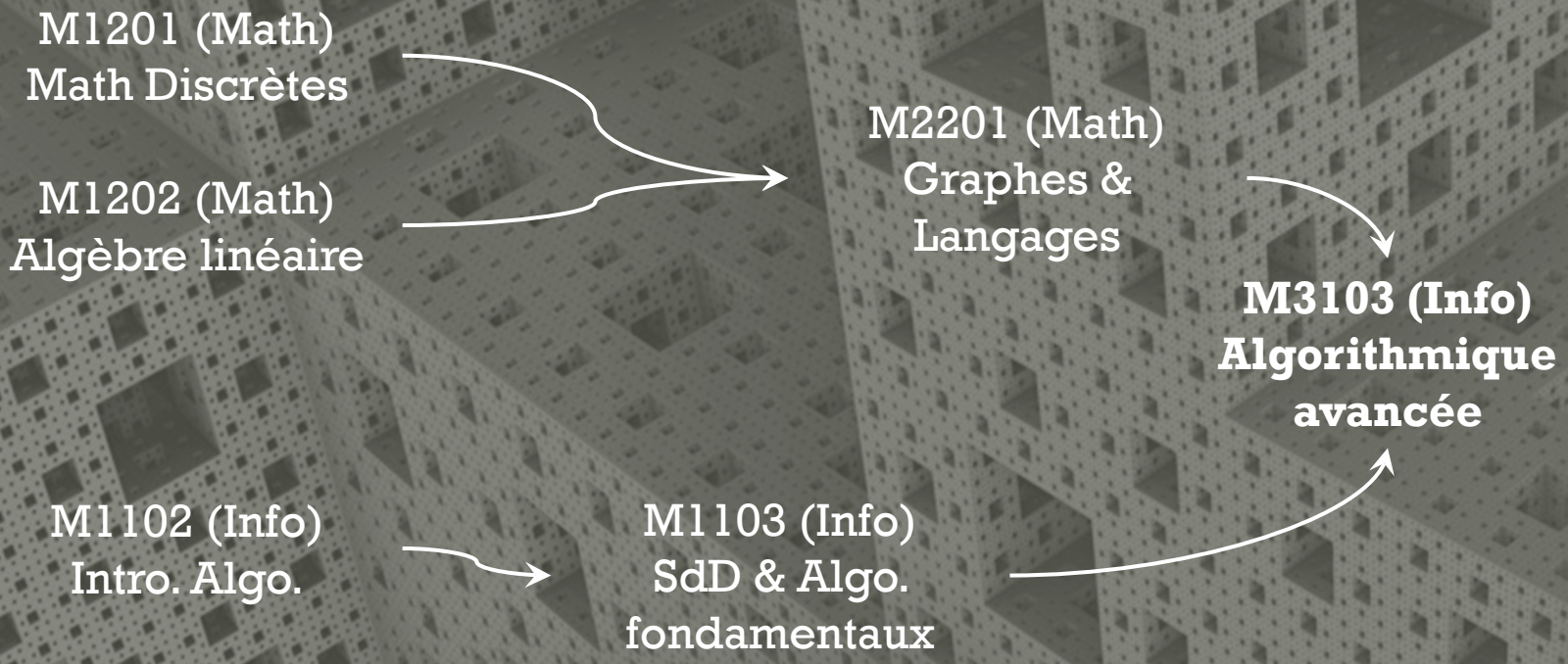
UNIVERSITÉ CÔTE D'AZUR
Université Nice Sophia Antipolis
IUT Nice Côte d'Azur

PPN 2013

http://www.iut-informatique.fr/docs/ppn/fr.pdf

- **Objectifs**
  - Utiliser & programmer des structures de données
  - Créer des algorithmes qui les manipulent
- **Contenu**
  - Structures de données **récursives**
  - Algorithmes **récursifs** et itératifs sur ces structures
  - Utilisation de structures de données **avancées**
- **Mise en œuvre**
  - Arbres, dictionnaires, ensembles …
- **Mots-clés**
  - Structures arborescentes, récursivité, structures associatives

Pré-requis



- **Considéré comme acquis !**
  - Algo de base (boucles, conditions, différences function / procédure)
  - Passage de paramètres (E, S, E/S)
  - Arbres & Graphes …

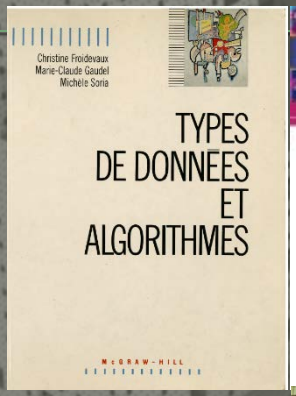
M313 à l'IUT de Nice Côte d'Azur

- **Objectifs**
  - Comprendre la récursivité
    - Visuelle / Jeux
  - Capable
    - D'écrire 1 Algo avancé
    - De vérifier 1 Algo (trace)
    - D'implémenter un Algo
    - De coupler SdD / Algo
  - Utiliser Langage Algorithmique
    - aucun langage de prog. !
  - TD/TP très longs !
- PPN = 8h CM, 10h TD, 12h TP
- Nice = 7h CM, 20h TD/TP
- Évaluation
  - DS (0.75), TP (0.75)

- **Plan**
  - Rappels
    - Fonction, procédure, paramètres, ALGORITHMES
  - Récursivité
    - Récurrence, Pile d'exécution, Types de récursivité, problème en récursif
  - Récursivité / Données
    - Puissance 4, Hanoi
    - Partitionnement, Tri Fusion, QuickSort
  - Récursivité / Structures
    - TAD Ensemble
      - Collection, Listes Chainées, Ensemble
    - TAD Arbre
      - Binaire, Équilibré, Syntaxe, B-Arbre, Arbre Coloré
    - TAD Graphe
      - Algo A*
  - Récursif vs Itératif

Mes sources

- **Web**
  - Florent Hivert, Algorithmique
  - Elise Bonzon, Algo. et structures, Récursion
  - Axel Chambily, Récursivité
  - François Lévy, Algo Avancé en IUT
  - Applet d'algos de tris
  - David, Galles, Visualisation d'Algorithmes
- **Papiers**
  - Algorithmes & structures de données génériques (cours & Exo en C), M. Divay
  - Types de données et algorithmes C. Froidevaux, M.C. Gaudel, M. Soria
  - Algorithmes – Notions de base , Cormen, 2013
    disponible à la BU-IUT
  - Algo et programmation Java, V. Granet, Dunod, 2010
  - Mastering Algorithms with C, O'Reilly, Kyle Oudon
- **MOOC**
  - https://www.programmation-recursive.net en Scheme
  - http://www.france-ioi.org/algo/

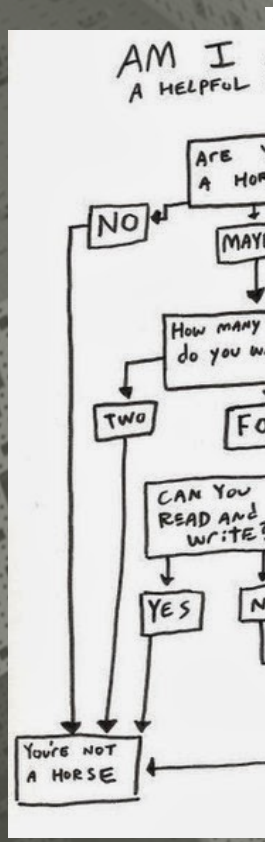


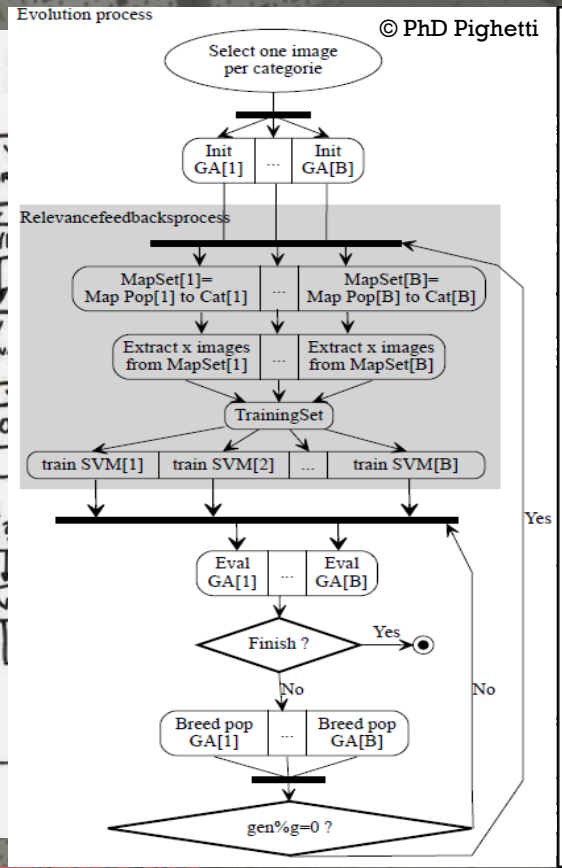
1 Rappels

Notion d'Algorithme / Organigramme
Fonction / Procédure
Passage de paramètres
Trace
…

Algorithme = Organisation de commandes

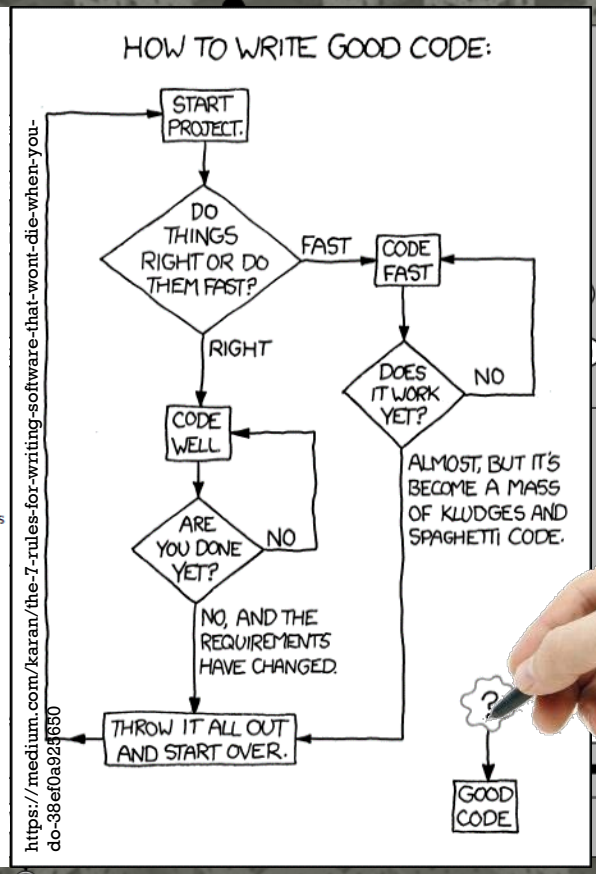
Organigramme

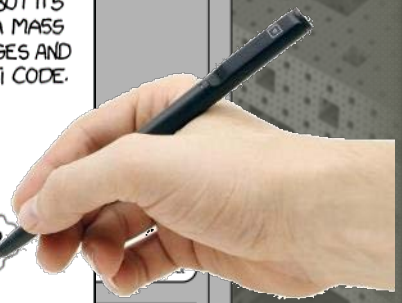




https://en.wikipedia.org/wiki/Flowchart

UML = Diagramme d'activité





https://en.wikipedia.org/wiki/Activity_diagram

Algorithmes = Pseudo-Code

https://en.wikipedia.org/wiki/Pseudocode

Algorithm 1: MAX (A)

In : Un ensemble fini de valeurs $A = \{a_1, a_2, \ldots, a_n\}$
Out :
In/Out:
Do : Donne la plus grande valeur de A

```
1 begin
2     max ← a_1
3     for i = 2 to taille(A) do
4         if a_i > max then
5             max ← a_i
6     return max
```

Algorithm 2: PGPABDE

In : A population P_t of λ solutions
Out :
In/Out:
Do : Return best solution found so far

```
1  begin
2      t ← 0; N_f ← 0; N_f̂ ← 0
3      archive ← ∅
4      P_t ← Initialize(size=λ)
5      Evaluate(P_t, f, c); N_f ← N_f + λ
6      archive ← {[x, f(x)], x ∈ P_t}
7      f̂ ← Build(archive)
8      while N_f < N_fmax and N_f̂ < N_f̂max do
9          G_t ← Generate(P_t)
10         if t%μ = 0 then
11             Evaluate(G_t, f, c); N_f ← N_f + λ
12             archive ← archive ∪{[x, f(x)], x ∈ G_t}
13             f̂ ← Build(archive)
14             P_t ← G_t
15         else
16             Evaluate(G_t, f_m, c); N_f̂ ← N_f̂ + λ
17             P_t ← {max(G_t^i, P_t^i), ∀i ∈ [0, len(G_t)]}
18         t ← t + 1
19     return solution ← x*/f(x*) > f(x), ∀x ∈ archive
```

- Différences / Similitudes ?
- Fonctions / Procédures ?
- Erreurs ?

Un peu de formalisme

- 2 types de méthodes

Procedure MarquerEnsemble($\mathcal{E}$: Liste< *Entier* >)

```
In     :
Out    :
In/Out: E index de ...
Do     : Expliquer ce que fait la procédure
1 begin
2     foreach e in E do
3         mark(e)
```

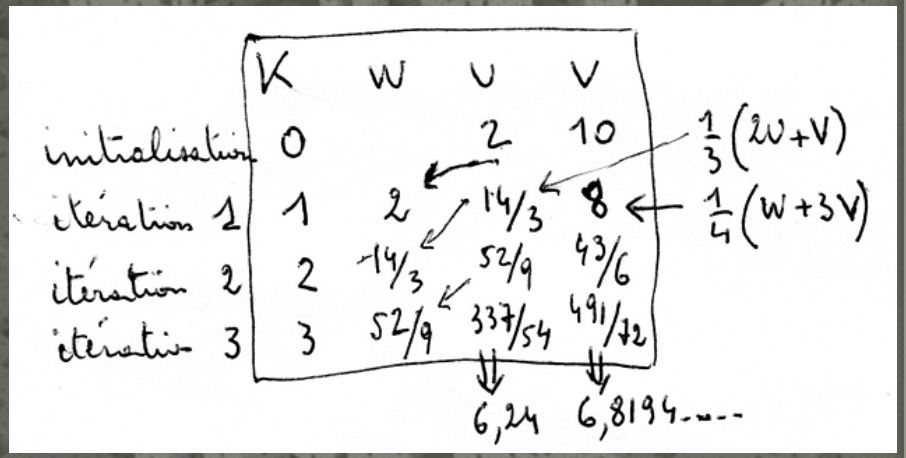
Function Taille($\mathcal{E}$: Liste< *Entier* >) : TypeDeRetour

```
In     : E index de ...
Out    :
In/Out: c'est possible !
Do     : Calcule le nb d'éléments contenus dans E
1 begin
2     nb ← 0
3     foreach e in E do
4         nb ← nb + 1
5     return nb
```

- Ingrédients
  - Conditions
    - If…Then…Else If …End, Switch…Case…End
  - Boucles
    - For, Foreach, Repeat…Until, While, Do… While
  - Appel de ss-prog

Trace d'Algo

- **Au départ**
  - Algo +
    Ensemble de test
    <Entrée, Résultat>
- **Sortie**
  - Tableau
    - Ligne = N°ligne algo
    - Colonne = Variables / Affichage
    - Case = valeur de la variable lors de l'exécution de la ligne
- **Ce qui est difficile ?**
  - Identifier les données caractéristiques en Entrée
  - Suivre SCRUPULEUSEMENT l'algo !



Un petit test

Procedure Echange(a, b : Réel)

In : a, b
Do : Échange valeur de a et b

```
1 begin
2 |  t ← a
3 |  a ← b
4 |  b ← t
```

Ne fonctionne que si a & b sont déclarés en E/S

Procedure TestEchange

Do : Test la procédure Echange

```
1 begin
2 |  x ← 5
3 |  y ← 2
4 |  Echange (x, y)
5 |  x = ?
```

Procedure EchangeOK(a, b : Réel)

In : ∅
In/Out: a, b
Do : Échange valeur de a et b

```
1 begin
2 |  t ← a
3 |  a ← b
4 |  b ← t
```

Passage par Référence / Adresse

- **Principe**
  - Adresse ou Référence à la variable transmise à la méthode
  - ⇒ Toute modification du paramètre est répercutée ds ss-prog appelant
  - Paramètre dit en ENTRÉE/SORTIE (E/S)
- **Intérêts ?**
  - Pour conserver modifications faites sur 1 variable ds méthode
  - Pour éviter de copier une grosse structure de données (tableau par ex.) même si tableau pas modifié
  - Pour que la méthode puisse retourner plusieurs valeurs simultanément
    - Unique valeur de retour + paramètre(s) en E/S
- **Matérialisé différemment suivant langage**
  - ByRef en Visual Basic
  - Ref en C#
  - */& (pointeur) en C/C++
  - N'existe pas en Java ! (pour les types scalaires : int, double…)
    - Solution possible : définir 1 tableau !

Nombre de Paramètres

LANGAGE C

- Données séparées des ss-prog.
  - `SdD Data`
  - `...`
  - `méthode estVide(...)`
- Déclaration
  - `Struct Data {…}`
  - `boolean estVide(Data d) { … }`
- Utilisation
  - `Data d = ...`
  - `If (estVide(d)) ...`

PROG. OBJETS (JAVA)

- Données & Méthodes regroupées

```
TAD {
    SdD Data
    méthode estVide(…) }
```

- Déclaration

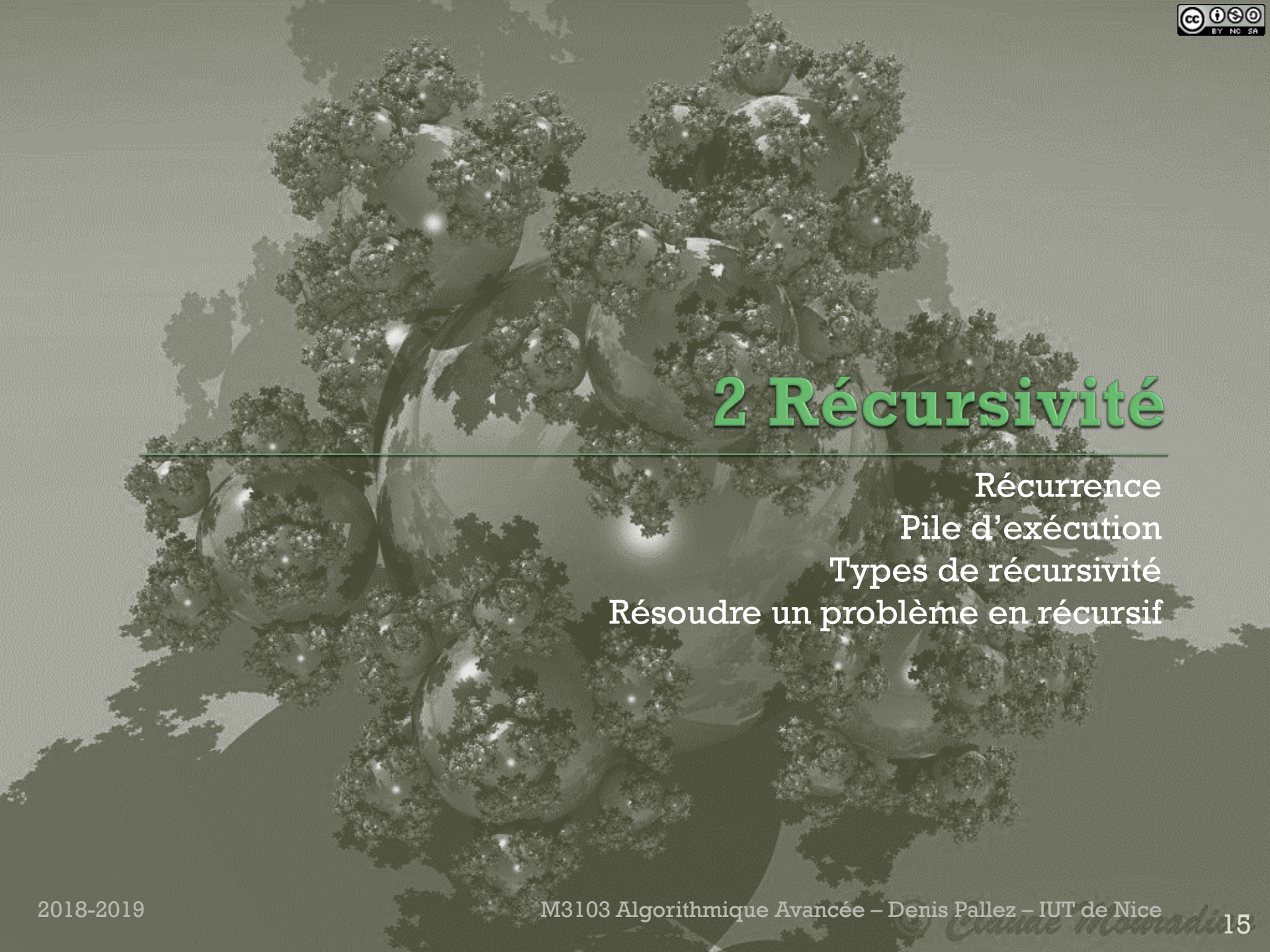
```
Class MyStruct {
    SdD Data;
    boolean estVide(){…}
}
```

- Utilisation

```
MyStruct d = new … ;
If (d.estVide()) …
```

Sources

- Damien Massé, Algo. récursifs : Applications
- Florent Hivert, Algorithmique
- Frédéric Fürst, Récursivité



2 Récursivité

Récurrence
Pile d'exécution
Types de récursivité
Résoudre un problème en récursif

Démonstration / Récurrence

- **Soit la proposition** S(n) : **la somme des n premiers entiers est égale à** $\frac{n(n+1)}{2}$
- **Pour** n=0, S(0)=0x(0+1)/2=0 ⇒ **S(0) vraie**
- **Supposons que** S(n) **vraie** ∀ n>0
- S(n+1) = 1 + 2 + … + n + n+1

S(n+1) = S(n) + n+1

$= \frac{n(n+1)}{2} + n + 1$

$= \frac{n(n+1)}{2} + \frac{2(n+1)}{2}$

$= \frac{n(n+1)+2(n+1)}{2}$

$= \frac{(n+2)(n+1)}{2}$

Récursivité

S(0)=0

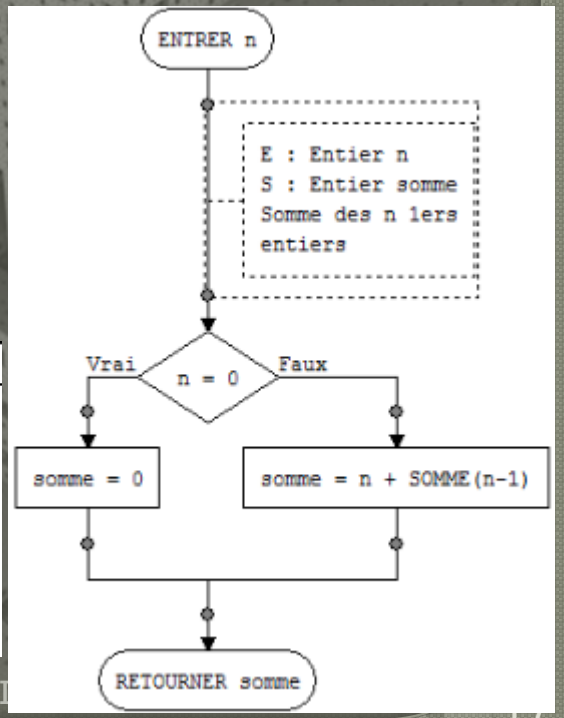
S(n)=n+S(n-1)

CQFD !

Récursivité

- **Idée**
  - Résoudre un problème complexe en résolvant le **même** problème avec des données plus simples, jusqu'à aboutir à des cas triviaux dont la solution est évidente
- **Méthode récursive**
  - Méthode qui s'appelle elle-même
    - S(0)=0
    - S(n)=n+S(n-1)

```
Function sommePremierEntiers(n : Entier) : Entier
  In      : n
  Do      : somme des n premiers entiers
1 begin
2   if n = 0 then
3     return 0
4   else
5     return n+sommePremierEntiers(n − 1)
```



De manière plus intuitive …

- **Problème de taille *n* : P(n)**
  - **Identifier la récurrence**
    - **Trouver un lien entre P(n) et P(n-1)**
  - **Identifier le(s) cas d'arrêt(s)**
    - **Identifier n_0 / $P(n_0)$ facile à calculer**
  - **Système calcule la solution S(n) à partir de $S(n_0)$**

$S(n)=n + S(n-1)$

$S(0)=0$

$\Rightarrow n_0=0, S(n_0)=0$

$P(n) \longrightarrow P(n-1) \longrightarrow \dots \longrightarrow P(n_0)$

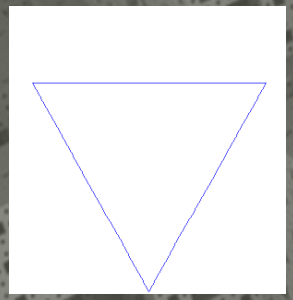
On sait calculer
AMORCE

$S(n) \longleftarrow S(n-1) \longleftarrow \dots \longleftarrow S(n_0)$

```
sommePremierEntiers(3) =
    3 + sommePremierEntiers(2) =
        2 + sommePremierEntiers(1) =
            1 + sommePremierEntiers(0) =
                0
            1
        3
    6
```

Exemples géométriques

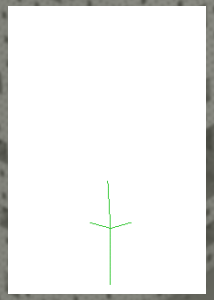
- Flocon de Koch



- Triangle de Sierpinski

- Éponge de Menger

Fougères



Chou romanesco

Comment ça fonctionne?

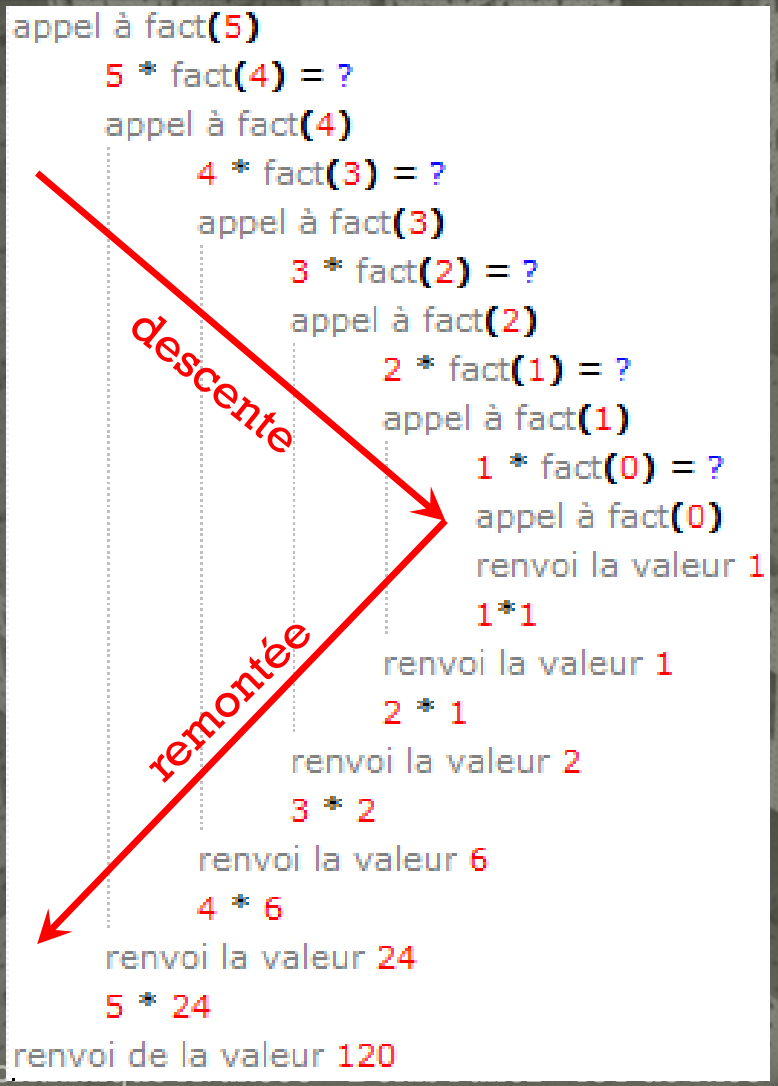
- $n! = 1 \times \ldots \times n$

```
Function Fact(n : Entier) : Entier
  In      : n
  Do      : calcul n!
1 begin
2 |  temp ← 1
3 |  for i = 2 to n do
4 |  |  temp ← temp × i
5 |  return temp
```

- $n! = n \times (n-1)!$

```
Function Fact(n : Entier) : Entier
  In      : n
  Do      : calcul n!
1 begin
2 |  if n < 2 then
3 |  |  return 1
4 |  else
5 |  |  return n× Fact(n − 1)
```

- **Et Fact (-1) ?**



Pile d'exécution

- **Définition**
  - Emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution
- **Principe LIFO – Last In, First Out**
  - Comme une pile d'assiettes !
- **Attention**
  - La pile possède une taille max.

  ⇒ trop de récursivité = dépassement de la taille max

  ⇒ « java.lang.StackOverflowError »

Cas d'arrêt

- **Cas simples**
  - Réponse connue sans appel récursif

Algorithm 3: MethodeRecursive(Parametres)

```
1 begin
2     if Condition_Arret then
3         Instructions0 // sans appel récursif
4     else
5         Instructions1
6         MethodeRecursive(Parametres_modifiés) // appel récursif
7         Instructions2
```

  - Il peut y en avoir plusieurs

Types de Récursivité

- **Récursivité Terminale**
  - Si aucun traitement effectué à la remontée d'un appel récursif (sauf retour d'une valeur)
    - La valeur retournée est directement la valeur obtenue par l'appel récursif sans opérations sur cette valeur
      ⇒ rien à retenir sur la pile
      ⇒ la fonction récursive est convertible en fonction itérative avec une boucle (cf. dérécursivation)
      - Certains compilateurs peuvent faire la transformation automatiquement : http://gcc.godbolt.org/

Types de Récursivité

- **Récursivité Non-Terminale**
  - Une méthode est récursive non-terminale si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus de la valeur de retour)

Terminale ou Non-Terminale ?

```
Function Fact(n : Entier) : Entier
  In      : n
  Do      : calcul n!
1 begin
2   if n = 0 then
3     return 1
4   else
5     return n× Fact(n − 1)
```

```
Function Fact(n, res : Entier) : Entier
  In      : n
  In/Out: res
  Do      : calcul n!
1 begin
2   if n = 0 then
3     return res
4   else
5     return Fact(n − 1, n × res)
```

- Les calculs se font à la remontée
  ⇒ Non-Terminale

- Appel avec res=1
  - Fact(3,1)
    - Fact(2,3)
      - Fact(1,6)
- Le calcul se fait à la descente
  ⇒ Terminale

Types de Récursivité

- **Récursivité Croisée**
  - 2 méthodes qui s'appellent l'une l'autre

```
Function estPair(n : Entier) : Booléen
   In      : n
1  begin
2      if n = 0 then
3          return True
4      else
5          return estImpair(n − 1)
6
```

```
Function estImpair(n : Entier) : Booléen
   In      : n
1  begin
2      if n = 0 then
3          return False
4      else
5          return estPair(n − 1)
6
```

Types de Récursivité

- **Récursivité Imbriquée**
  - Faire un appel récursif à l'intérieur d'un autre appel récursif
  - Exemple : Ackerman

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1,A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0. \end{cases}$$