



# Cours d'algorithmique

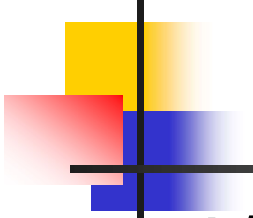
---



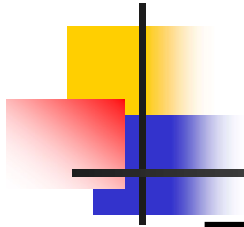
**ALGO**

**ALGO**

# Introduction

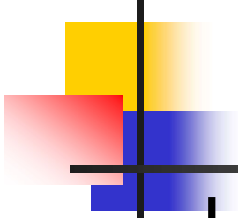
- 
- L'objectif de ce cours est de vous permettre :
    - D'acquérir la logique informatique pour résoudre des problèmes.
    - D'acquérir la démarche pour le raisonnement permettant de traiter les problèmes et de proposer une solution automatisable, exprimée au moyen de ce qu'on appelle algorithmes.

# Introduction

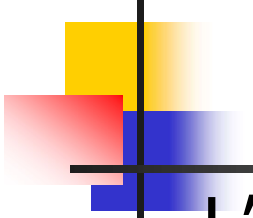


- Toute organisation possède un SI (Système d'Information) qui constitue son patrimoine en terme de données et procédures de traitement de ces données.
- L'informatique est la science des données qui est utilisée pour optimiser les SI
  - Elle permet de définir la façon de représenter, manipuler et traiter les données appartenant à un SI d'une organisation afin de mieux les exploiter.

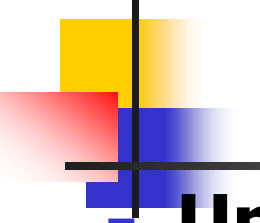
# Introduction

- 
- Le métier de l'informatique est l'informatisation totale ou partielle de procédures/processus des SI pour optimiser (simplifier/accélérer) leur exécution.
    - afin de faciliter le travail les utilisateurs.
  - L'informatisation d'une procédure/processus consiste à mettre en œuvre un programme (en utilisant un langage de programmation ex. Java, C#, ...) qui, une fois exécuté, produit le résultat attendu.

# Introduction

- 
- L'écriture de programmes informatiques passe par une étape de conception (de la solution)
    - Lors de cette étape, les programmes sont conçus sous forme **d'algorithmes**
  - **Un algorithme** exprime un raisonnement logique/mathématique pour **solutionner un problème donné**
    - L'algorithmique est l'activité d'exprimer les instructions résolvant un problème donné **indépendamment de tel ou tel langage** de programmation

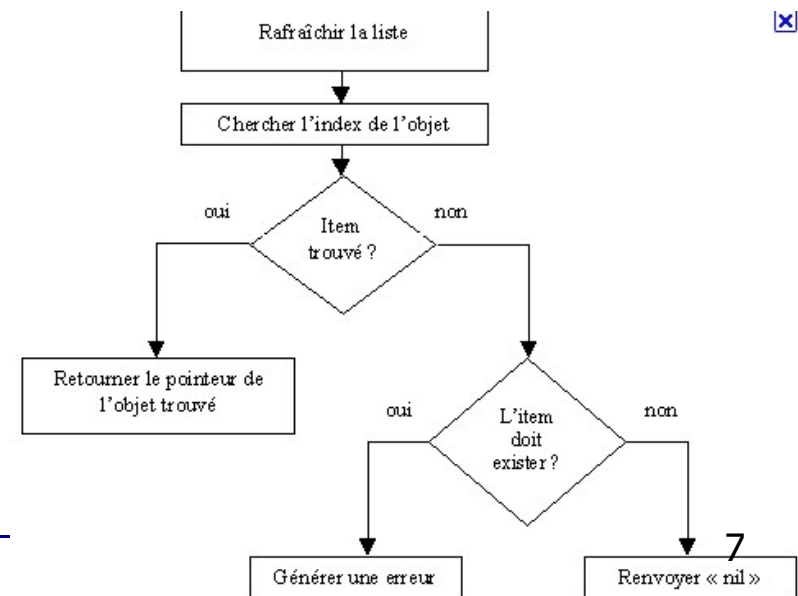
# Introduction

- 
- **Un algorithme est donc, une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné**
  - **Pour fonctionner, un algorithme doit contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter :**
    - **Le coder**
    - **Le vérifier**
    - **...**

# Conventions d'écriture

- **Un algorithme est écrit en pseudo-code**
  - **Pseudo-code** : ressemble à un langage de programmation authentique (C, Pascal, Java, ...) avec beaucoup de simplification.
    - Généralement dans une langue de communication ex. Français

NB : Il est possible d'écrire un algorithme au moyen d'**organigrammes** : représentation graphique avec des carrés, des losanges, etc.





# Ingrédients d'algorithmes

---

- Pour écrire un algorithme on utilise des :
  - Variables
  - Expressions & opérations
  - Bloc d'instructions
    - Affectation
    - Opérations
    - Fonctions (blocs de code nommés) et Appels de fonctions
  - Structures de contrôle :
    - Conditions,
    - Boucles,
    - ...





# Les Variables

---

- Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des **valeurs** de types différents :
  - ➔ Utilisation de **variables**
- Une variable est une **boîte** identifiée par un **nom**, dans laquelle on **stocke/retrouve/restocke** des **valeurs**



# Déclaration des variables

---

- C'est mentionner l'intention d'utiliser une variable en l'associant le cas échéant à un type *et l'initialiser, optionnellement, avec une valeur*
- Le **nom** de la variable obéit à des impératifs des identifiants des langages de programmation :
  - C'est une suite alpha-numérique et \_ ne commençant pas par un chiffre

*NB : certains langages comme XML autorisent d'autres caractères.*



# Déclaration des variables

---

- Le **type** permet de déterminer la taille et appliquer les contrôles nécessaires :
  - Types numériques : Entier, Réel, ...
  - Type caractères : 'a', "ab2 cdef5"
  - Type booléen : vrai, faux
- Syntaxe pour déclarer une variable :  
**Type** *nomVariable*      **ou bien** *nomVariable* **Type**
  - Exemples  
Entier x      ou bien      x entier  
Chaine de caractères str      str chaîne de caractères



# L'instruction d'affectation

---

- Affecter une variable c'est lui attribuer une valeur
  - Symbole utilisé  $\leftarrow$
  - Comptabilité entre le contenant et le contenu
  - De préférence, une variable assignée doit être au préalable déclarée

Exemple 1 :

$i \leftarrow 24$

- *Attribue la valeur 24 à la variable  $i$*

- Exemple 2 : affectation lors de la déclaration (initialisation)  
Booleen bool  $\leftarrow$  vrai



# Commentaires

---

- Texte libre (non-codable) qui explique des parties de l'algorithme

`/* Voici un commentaire qui n'est pas  
destiner à être codé */`



# Exercice

---

- **Exercice**

- Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

**Début**

**Entier** A, B, C

A ← 5

B ← 3

C ← A + B

A ← 2

C ← B - A

**Fin**



# Corrigé

---

■ Après	La valeur des variables est :		
$A \leftarrow 5$	$A = 5$	$B = ?$	$C = ?$
$B \leftarrow 3$	$A = 5$	$B = 3$	$C = ?$
$C \leftarrow A + B$	$A = 5$	$B = 3$	$C = 8$
$A \leftarrow 2$	$A = 2$	$B = 3$	$C = 8$
$C \leftarrow B - A$	<b><math>A = 2</math></b>	<b><math>B = 3</math></b>	<b><math>C = 1</math></b>



# Exercice

---

- **Exercice 2**
- Un classique absolu, qu'il faut absolument maîtriser :
  - Ecrire un algorithme permettant d'échanger (permuter) les valeurs de 2 variables A et B, et ce quelque soit leur contenu préalable.





# Corrigé

---

**Début**

...

$C \leftarrow A$

$A \leftarrow B$

$B \leftarrow C$

**Fin**

- On est obligé de passer par une variable dite temporaire (la variable C).



# Expressions et opérateurs

---

- Une **expression** est un ensemble d'opérandes, reliées par des **opérateurs**, et équivalent (évaluable) à une seule valeur
- Exemples d'opérateurs :  
 **$+$ ,  $*$ , **ET**, ...**
- Exemples d'expressions :  
**7**  
**5+4**  
**123-45+844**  
**oto-12+5-Riri**



# Opérateurs

- Opérateurs arithmétiques

Opérateur	Signification	<i>exemple</i>	<i>résultat</i>
+	addition	$1 + 2$	3
-	soustraction	$5 - 3$	2
*	multiplication	$3 * 5$	15
/	division	$15 / 3$	5



# Opérateurs

- Opérateurs de comparaison

Opérateur	Signification	<i>exemple</i>	<i>résultat</i>
$>$	supérieur	$5,1 > 2$	VRAI
$<$	inférieur	$3,1 < 9$	VRAI
$\leq$	supérieur ou égal	$5 \leq 2,1$	FAUX
$\geq$	inférieur ou égal	$3 \geq 9$	FAUX
$=$	égal	$3,3 = 3,3$	VRAI
$\neq$	différent	$3,3 \neq 3,3$	FAUX

# Opérateurs

## Opérateurs logiques (booléens)

Opérateur	<i>exemple</i>	<i>résultat</i>
et	VRAI et VRAI VRAI et FAUX FAUX et FAUX	VRAI FAUX FAUX
ou	VRAI ou VRAI VRAI ou FAUX FAUX ou FAUX	VRAI VRAI FAUX
non	non VRAI non FAUX	FAUX VRAI

A	B	A ou B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ET B
0	0	0
0	1	0
1	0	0
1	1	1

A	NON A
0	1
1	0



# Opérateurs

---

- Opérateur alphanumérique : + (ou bien &)

Début

Chaines de caractères A, B, C

A ← "Algo"

B ← "rithme"

C ← A + B

/\* C contiendra "Algorithme" \*/

Fin



# Instructions

---

- Une ***instruction*** est un ordre composé d'une ou plusieurs expressions
  - Affectation par une valeur ou une expression
    - $x \leftarrow 5$                     / \* instruction d'affectation de x avec une valeur 5 \*/
    - $y \leftarrow x*2+1$             / \* instruction d'affectation de y avec une expression évaluée à 11 \*/
  - Expression de calcul
    - $x+y$
  - Appel de fonction
  - ....
- ***Un bloc d'instructions*** est formé d'une ou plusieurs instructions pouvant être exécutées conjointement

# Fonctions / Appel de fonctions

- **Une fonction est un bloc d'instructions nommé**

- défini une fois
- exécuté à chaque besoin, par appel

Détaillée plus loin

- **Sortie : Ecrire**

- Ex.    Ecrire ("Saisissez votre nom" )  
          Ecrire (x)            /\* x est une variable \*/

Appel de la fonction Ecrire. Le texte/le contenu de la variable entre parenthèses est affiché

- **Entrée : Lire**

- Ex. Lire (x)                    /\* x est une variable \*/

Appel de la fonction Lire. Lit ce qui est saisi en entrée (clavier) et le met dans la variable entre parenthèses

- **Autres :**

- Racine\_carree (x)
- Majuscule(s)
- ...



# Structures de contrôle :



## Si

- Si

<b>Si</b> (condition) <b>Alors</b> <i>...action1...</i> <b>FinSi</b>	<b>Si</b> (x=3) <b>Alors</b> <i>Ecrire ("x vaut 3") ;</i> <b>FinSi</b>
--	--

- Si-Sinon

<b>Si</b> (condition) <b>Alors</b> <i>...action1...</i> <b>Sinon</b> <i>...action 2...</i> <b>FinSi</b>	<b>Entier x</b> $\leftarrow$ 0 ; <b>Si</b> (x>0) <b>Alors</b> <i>Ecrire ("x positif") ;</i> <b>Sinon</b> <i>Ecrire ("x non positif") ;</i> <b>FinSi</b>
---	--

# Structures de contrôle :



## Si

---

- **Exercice**

- Ecrire un algorithme qui lit 3 valeurs et affiche la plus petite des trois

# Structures de contrôle :

## TantQue

- Boucle Tant que

Forme 1

**TantQue** (condition) faire  
    *...action1...*  
**FinTantQue**

**Entier**  $x \leftarrow 0$  ;  
**TantQue** ( $x \leq 10$ ) faire  
     $x \leftarrow x + 1$  ;  
**FinTantQue**

Forme 2

**Faire**  
    *...action1...*  
**TantQue** (condition)

**Entier**  $x \leftarrow 0$  ;  
**Faire**  
     $x \leftarrow x + 1$  ;  
**TantQue** ( $x \leq 10$ )



# Structures de contrôle :

## TantQue

---

- **Exercice**

- Ecrire un algorithme qui lit un entier et affiche le factoriel de cet entier

# Structures de contrôle :

## Répéter

- Boucle Répéter

**Répéter**

*...action1...*

**Jusqu'à** (condition)

**Entier  $x \leftarrow 0$  ;**

**Répéter**

*$x \leftarrow x + 1$  ;*

**Jusqu'à** ( $x=10$ )



# Structures de contrôle :

## Répéter

---

- **Exercice**

- Ecrire un algorithme qui lit un entier et affiche le factoriel de cet entier

# Structures de contrôle :

## Pour

- Boucle Pour

**Pour** variable  $\leftarrow$  valeur 1 **A**  
valeur 2 **Faire**  
    *...action1...*  
**Fin Pour**

**Pour**  $i \leftarrow 1$  **A** 10 **Faire**  
    afficher( $i * i$ );  
**Fin Pour**

Pas, par défaut, 1

**Pour** variable  $\leftarrow$  valeur 1 **A**  
valeur 2 **Pas** valeur 3 **Faire**  
    *...action1...*  
**Fin Pour**

**Pour**  $i \leftarrow 1$  **A** 10 **Pas** 2  
**Faire**  
    afficher( $i * i$ );  
**Fin Pour**

Si le Pas est négatif,  
valeur 1 doit être  $\geq$  valeur 2



# Structures de contrôle :

## Pour

---

- **Exercice**

- Ecrire un algorithme qui lit un entier et affiche le factoriel de cet entier





# Fonctions & procédures

- A quoi ça sert ?
  - Avoir un bloc de traitement à appeler selon les besoins
- Une fonction est composée :
  - **Nom (identifiant)**
  - de **paramètre(s)** (ou argument)
  - du **type** de la variable renvoyée par la fonction
    - une procédure ne renvoie pas de retour
  - du **corps** de la fonction

**Fonction** *NomFonction* (*var1,...,varN*)

**Retourne:** *type*

*...corps de la fonction...*

**Retourner** *var* ;

**FinFonction**

**Fonction** *auCarre* (*Entier x*)

**Retourne:** *Entier*

$x \leftarrow x * x ;$

**Retourner** *x* ;

**FinFonction**



# Fonctions & procédures

- Dans certains langages de programmation on distingue entre fonction et procédure :
  - Une fonction renvoie une valeur de retour
  - Une procédure (routine) ne renvoie pas de valeur de retour
    - La procédure peut utiliser les paramètres pour transmettre des résultats

**Procedure** *nomProcedure* (*var1,...,varN*)

*...corps de la procedure...*

**FinProcedure**

Entier y

**Procedure** *auCarre* (*Entier x*)

$y \leftarrow x * x;$

**FinProcedure**

- Dans certains langages de programmation une procédure est une fonction sans retour



# Fonctions & procédures

---

- Appel d'une fonction :
  - *NomFonction* (*valeurVar1*, ... *valeurVarN*)
  - Ou bien
  - *variable*  $\leftarrow$  *NomFonction* (*valeurVar1*, ... *valeurVarN*)

## Exemples :

*auCarre* (*4*) ;

Entier *z*  $\leftarrow$  *auCarre* (*4*) ;    //ici z=16

- On peut utiliser des noms de fonctions des langages de programmation ou leur traduction : *split*, *startWidth*, *uppercase*, *indexOf*, ....



# Fonctions & procédures

- Une fonction / procédure peut soulever des erreurs au moyen d'un mécanisme dit 'Exception'

Fonction factoriel (Entier n) Retourne Entier

Si ( $n < 0$ ) Alors

Exception("Pas de factoriel")

FinSi

...

...

....

Fin Fonction

Début

Entier n, f

Lire (n)

$f \leftarrow \text{factoriel}(n)$

Si (exception) Alors

Afficher (exception)

Sinon

Afficher ("Factoriel de " + n + " = "+f)

FinSI

Fin



# Fonctions & procédures

---

## ■ Exercice

- Ecrire un algorithme pour une fonction **factoriel** qui prend comme argument un entier et retourne le factoriel de cet entier.
- Ecrire un algorithme qui lit un entier, calcul le factoriel de cet entier en appelant la fonction **Factoriel** et affiche le résultat retourné par la fonction.

Rappel :       $0! = 1$

$1! = 1$

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

Pas de factoriel pour les négatives

# Fonctions & procédures

## Corrigé

Fonction factoriel (Entier n) Retourne Entier

**Entier i**  $\leftarrow$  1, **f**  $\leftarrow$  1

Si (n < 0) Alors

    Exception("Pas de factoriel")

FinSi

**TantQue (i  $\leq$  n) Faire**

**f**  $\leftarrow$  **f** \* **i**

**i**  $\leftarrow$  **i** + 1

**FinTantQue**

**Retourner f**

Fin Fonction

Début

Entier n, f

Lire (n)

f  $\leftarrow$  factoriel(n)

Si (exception) Alors

    Afficher (exception)

Sinon

    Afficher ("Factoriel de " + n + "=" + f)

FinSI

Fin

# Structures de données

## Tableaux

### Tableau

- Variable pouvant stocker plusieurs valeurs en même temps
- Possède une taille = nombre des valeurs pouvant être stockées
- Les valeurs sont toutes du même type
- Chaque valeur est accessible via un indice allant de 0 à taille -1
- Pour un tableau tab, sa taille est obtenue par **tab→Taille**

### ■ Exemple :

Entier tab[10]

Entier i ← 3

tab[0] ← 15    /\* Affecte 15 dans la cellule 0 du tableau) \*/

Lire (tab[i])    /\* Lit la cellule 3 (i vaut 3) \*/

tab [i] ← tab [i] x 2 / Affecte la cellule 3 avec le double de sa valeur 15x2 = 30 \*/

Pour i ← 0 à tab→Taille – 1 Faire    /\* Affiche tous les éléments du tableau un par un \*/

    Afficher (tab[i])

Fin Pour

Valeur	45	154	58	78	31	5	74
Index	0	1	2	3	4	5	6

PS : Il est possible d'utiliser plusieurs dimensions : 2 (matrice), 3, ...

# Structures de données

## Tableaux



---

### ■ **Exercice 1**

- Ecrire un algo qui :
  - Définit un tableau de 10 entiers
  - Lit (par boucle) dans le tableau
  - Calcule la somme des éléments du tableau



# Structures de données

## Tableaux

### ■ Corrigé de l'exercice 1

Début

Entier  $t[10]$ ,  $i$ ,  $somme \leftarrow 0$

Pour  $i \leftarrow 0$  à 9 Faire

    Lire( $t[i]$ )

FinPour

Pour  $i \leftarrow 0$  à  $t \rightarrow \text{Taille}-1$  Faire

$somme \leftarrow somme + t[i]$

FinPour

Afficher ( $somme$ )

Fin

Ecrire un algo qui :

- Définit un tableau de 10 entiers
- Lit (par boucle) dans le tableau
- Calcule la somme des éléments du tableau

# Structures de données

## Tableaux



---

### ■ **Exercice 2**

- Ecrire un algo qui :
  - Définit un tableau de 10 entiers
  - Lit (par boucle) dans le tableau
  - Détermine le min parmi les éléments dans le tableau

# Structures de données

## Tableaux

### ■ Corrigé de l'exercice 2

Début

Entier t[10], i, min

Pour i  $\leftarrow$  0 à t  $\rightarrow$  Taille-1 Faire

    Lire(t[i])

FinPour

min  $\leftarrow$  t[0]

Pour i  $\leftarrow$  1 à t  $\rightarrow$  Taille-1 Faire

    Si (min > t[i]) Alors

        min  $\leftarrow$  t[i]

    FinSI

FinPour

Afficher (min)

Fin

Ecrire un algo qui :

- Définit un tableau de 10 entiers
- Lit (par boucle) dans le tableau
- Détermine le min parmi les éléments dans le tableau

# Structures de données

## Tableaux et fonctions

### ■ Exercice 3 : Tableau en paramètre de fonction

- Compléter la fonction ***produit*** qui :
  - Calcule et renvoie le produit des éléments dans le tableau passé en paramètre

Fonction ***produit*** (Entier *tableau*[]) Retourne Entier

Entier *taille\_tableau* ← *tableau* → Taille

Entier *pdt*

....

....

Retourner *pdt*

FinFonction

Permet de récupérer la  
taille du tableau

# Structures de données

## Tableaux associatifs

- Tableau associatif : dictionnaire
  - Variable pouvant stocker plusieurs valeurs en même temps
  - Chaque valeur est accessible via une clé (chaîne de caractères)

- Exemple :

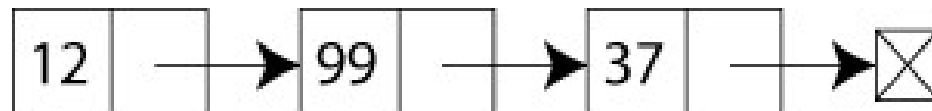
```
Dictionnaire tab;  
tab["un"] ← 1  
tab["dix"] ← "blanc"
```

```
Afficher (tab["un"] )
```

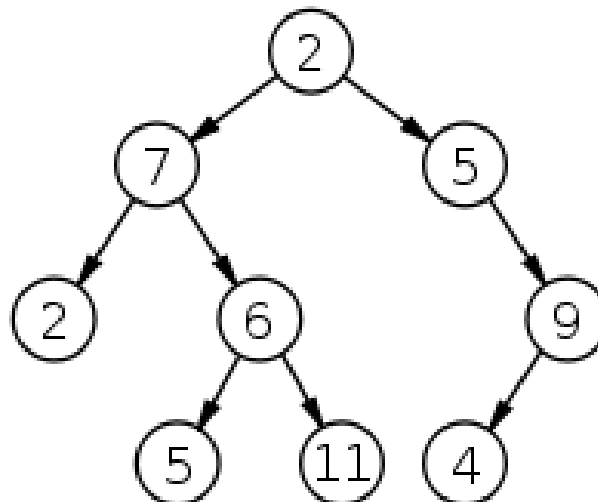
# Autres structures de données

## Listes & arbres

- Liste



- Arbre





# Fonctions récursives

- Fonction récursive = fonction qui fait appel à elle même
- Autre forme de boucler
- Convient pour les traitements sur des structures récursives comme les arbres
- Exemple :

Fonction Somme(Entier n) retourne Entier

Si  $n = 0$  alors

Retourner 0

FinSi

Retourner  $n + \text{Somme}(n-1)$

Fin Fonction



# Fonctions récursives

---

## ■ Exercice

- Ecrire un algorithme pour une fonction **récursive factoriel** qui prend comme argument un entier et retourne le factoriel de cet entier.
- Ecrire un algorithme qui lit un entier, calcul le factoriel de cet entier en appelant la fonction **Factoriel** et affiche le résultat retourné par la fonction.



# Fonctions récursives

## Corrigé

Fonction factoriel (Entier n) Retourne Entier

Si ( $n < 0$ ) Alors  
    Exception("Pas de factoriel")

Finsi

Si ( $n \leq 1$ ) Alors  
    Retourner 1

FinSi

Retourner  $n * \text{factoriel}(n-1)$

Fin Fonction

Début

Entier n, f

Lire (n)

$f \leftarrow \text{factoriel}(n)$

Si (Exception) Alors

    Afficher (Exception)

Sinon

    Afficher ("Factoriel de " + n + "=" + f)

FinSi

Fin

# Fonctions récursives

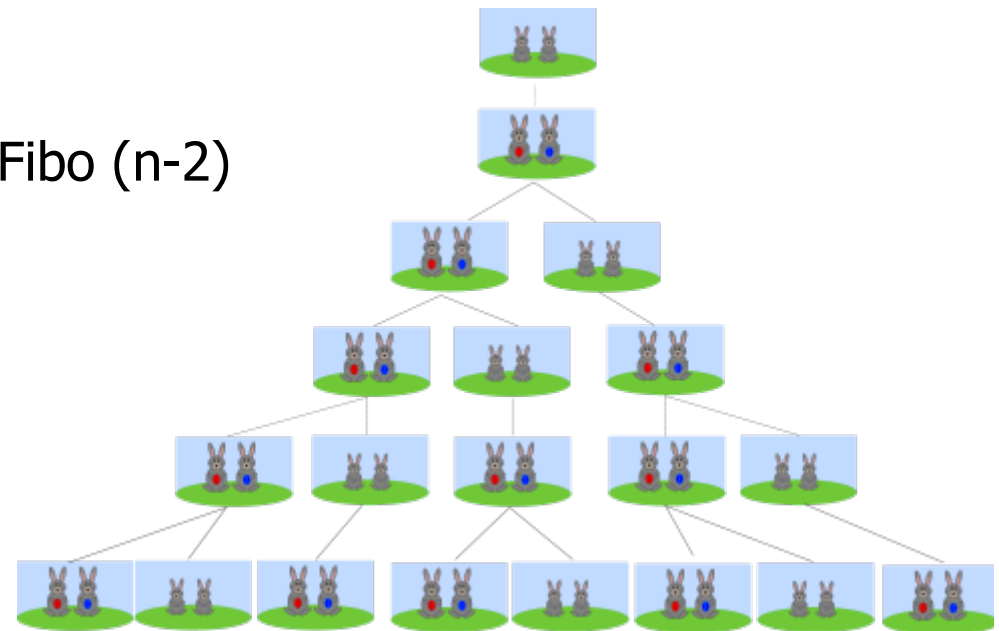
## ■ Exercice

- Sur le même principe de la fonction Factoriel, écrivez un algorithme pour la fonction Fibonacci

Fibo (0) = 0

Fibo (1) = 1

Fibo (n) = Fibo (n-1) + Fibo (n-2)



$\mathcal{F}_0$	$\mathcal{F}_1$	$\mathcal{F}_2$	$\mathcal{F}_3$	$\mathcal{F}_4$	$\mathcal{F}_5$	$\mathcal{F}_6$	$\mathcal{F}_7$	$\mathcal{F}_8$	$\mathcal{F}_9$	$\mathcal{F}_{10}$	$\mathcal{F}_{11}$	$\mathcal{F}_{12}$	$\mathcal{F}_{13}$	$\mathcal{F}_{14}$	$\mathcal{F}_{15}$	$\mathcal{F}_{16}$	$\mathcal{F}_{17}$	$\mathcal{F}_{18}$	$\mathcal{F}_{19}$	$\mathcal{F}_{20}$	$\mathcal{F}_n$
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765	$\mathcal{F}_{n-1} + \mathcal{F}_{n-2}$

# Fonctions récursives

## Corrigé

```
Fonction Fibbo(Entier n) retourne Entier
    Si (n < 0) Alors
        Exception("Valeur positive requise")
    FinSi
    Si (n ≤ 1) Alors
        Retourner n
    FinSi
    Retourner Fibbo (n-1)+Fibbo(n-2)
Fin Fonction
```

```
Début
    Entier n, f
    Lire (n)
    f ← Fibbo(n)
    Si (Exception) Alors
        Afficher (Exception)
    Sinon
        Afficher ("Fibbo de " +n + "=" +f)
    FinSi
Fin
```

# Recherche dans des tableaux



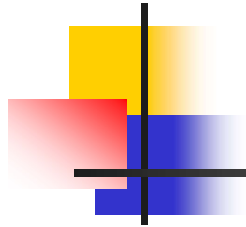
## **Tableau non trié** (*recherche aveugle*)

- Ecrire une fonction Rechercher qui effectue une recherche dans un tableau d'une valeur passée en paramètre et renvoie son index si la valeur est trouvée dans le tableau ou -1 sinon.

Fonction rechercher(Entier tab[], Entier v) : Retourne Entier

Fin Fonction

# Recherche dans des tableaux



## Tableau non trié – Corrigé 1

Fonction rechercher(Entier tab[], Entier v) : Retourne Entier

Entier i  $\leftarrow$  0

Entier taille\_tab  $\leftarrow$  tab->Taille

TantQue (i < taille\_tab ET tab[i]  $\neq$  v) Faire

    i  $\leftarrow$  i + 1

FinTantQue

Si (i < taille\_tab) Alors

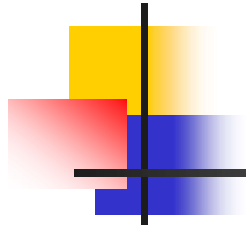
    Retourner i

FinSi

Retourner -1

Fin Fonction

# Recherche dans des tableaux



## Tableau non trié – Corrigé 2

Autre version :

Fonction rechercher(Entier tab[], Entier v) : Retourne Entier

Entier i  $\leftarrow$  0

Entier taille\_tab  $\leftarrow$  tab  $\rightarrow$  Taille

Pour (i  $\leftarrow$  0 A taille\_tab – 1)) Faire

Si (tab[i] = v) Alors

Retourner i

FinSi

FinPour

Retourner -1

Fin Fonction

# Recherche dans des tableaux

## Par hachage de tableau trié (*recherche guidée*)

- Ecrire une fonction Rechercher qui effectue une recherche dans un tableau d'une valeur passée en paramètre et renvoie son index si la valeur est trouvée dans le tableau ou -1 sinon.
- La fonction utilise le principe de hachage :
  - tableau scindé en 2 parties, on ne considère que la partie susceptible de contenir la valeur recherchée,
  - La moitié retenue est à son tour scindée en 2 parties, ...

Fonction rechercher(Entier tab[], Entier borneInf, Entier borneSup, Entier v) : Retourne Entier

Fin Fonction

# Recherche dans des tableaux

## Par hachage de tableau trié - Corrigé

```
Fonction rechercher(Entier tab[], Entier borneInf, Entier borneSup, Entier v) : Retourne Entier
Entier i
Si (borneSup < borneInf OU tab[borneSup] < v OU tab[borneInf] > v ) Alors
    Retourner -1
FinSi
Si (tab[borneSup] = v) Alors
    Retourner borneSup
FinSi
Si (tab[borneInf] = v ) Alors
    Retourner borneInf
FinSi
i ← (borneSup + borneInf)/2
Si (tab[i] = v) Alors
    Retourner i
FinSi
Si (tab[i] > v) Alors
    Retourner Rechercher(tab, borneInf+1, i-1, v)
FinSi
Retourner Rechercher (tab, i+1, borneSup-1, v)
```

1<sup>er</sup> appel

....

Entier idx

idx ← rechercher(tab, 0, tab→taille - 1, v)





# Tri des tableaux

## ■ Par sélection

- Rechercher le plus grand élément du tableau pour le placer à la fin du tableau,
- Puis de chercher le plus grand élément dans le reste (en excluant le dernier élément) et de le mettre en avant dernier, etc...
  - On suppose que le premier/dernier élément du tableau considéré est le plus grand (en mémorisant son indice dans une variable)
  - On parcourt le tableau considéré pour chercher si un élément est plus grand que lui
  - Si c'est le cas on met à jour la variable indice
  - A la fin du parcours si l'indice est différent de celui de la fin du tableau considéré, on permute les deux contenus et on réduit la taille du tableau restant à trier de 1



# Tri des tableaux

## ■ Par sélection

### ■ Indication de Solution

```
Fonction TriSelection (Entier tableau[]) Retourne Entier[]  
  Entier taille_tableau ← tableau→Taille  
  ...  
  Pour i ← taille_tableau -1 A 1 Pas -1 Faire  
    ....  
  FinPour  
  Retourner tableau  
FinFonction
```



# Tri des tableaux

## ■ Par sélection

### ■ Solution

```
Fonction TriSelection (Entier tableau[]) Retourne Entier[]  
  Entier i, i_max, j, temp  
  Entier taille_tableau ← tableau → Taille  
  Pour i ← taille_tableau - 1 A 1 Pas -1 Faire  
    i_max ← i  
    Pour j ← 1 A i Faire  
      Si (tableau[j] > tableau[i_max]) Alors  
        i_max ← j  
      FinSi  
    FinPour  
    Si (i_max ≠ i) Alors  
      temp ← tableau[i]  
      tableau[i] ← tableau[i_max]  
      tableau[i_max] ← temp  
    FinSi  
  FinPour  
  Retourner tableau  
FinFonction
```



# Tri des tableaux

## ■ Par sélection

### ■ Solution 2

```
Fonction getMaxIdx(Entier tableau[], Entier borne) retourne Entier
    i_max ← 0
    Pour j ← 1 A borne Faire
        Si (tableau[j] > tableau[i_max]) Alors
            i_max ← j
        FinSi
    FinPour
    Retourner i_max
Fin Fonction
```

```
Fonction TriSelection (Entier tableau[]) Retourne Entier[]
    Entier i, i_max, j, temp
    Entier taille_tableau ← tableau→Taille
    Pour i ← taille_tableau -1 A 1 Pas -1 Faire
        i_max ← getMaxIdx(tableau, i)
        Si (i_max ≠ i) Alors
            temp ← tableau[i]
            tableau[i] ← tableau[i_max]
            tableau[i_max] ← temp
        FinSi
    FinPour
Fin Fonction
```

# Tri des tableaux

## ■ Par sélection

- Autre solution récursive

Début

Entier tab[10], tab2[10], i  
Pour i  $\leftarrow$  0 à tab  $\rightarrow$  Taille Faire  
    Lire (tab[i])

FinPour

tab2  $\leftarrow$  Trier (tab, tab  $\rightarrow$  Taille)

Fin

Fonction Trier(Entier tab[], Entier taille\_tableau)  
Retourne Entier []

Entier j, i\_max, temp

Si (taille\_tableau  $\leq$  1) Alors

    Retourner tab

FinSi

i\_max  $\leftarrow$  0

Pour j  $\leftarrow$  1 A taille\_tableau -1 Faire

    Si (tableau[j] > tableau[i\_max]) Alors

        i\_max  $\leftarrow$  j

    FinSi

FinPour

Si (i\_max  $\neq$  taille\_tableau-1) Alors

    temp  $\leftarrow$  tableau[taille\_tableau-1]

    tableau[taille\_tableau-1]  $\leftarrow$  tableau[i\_max]

    tableau[i\_max]  $\leftarrow$  temp

FinSi

Retourner Trier(tab, taille\_tableau -1)

Fin Fonction



# Tri des tableaux

## ■ Par bulle

- Parcourt le tableau, et compare les couples d'éléments successifs.
- Lorsque deux éléments successifs ne sont pas dans l'ordre croissant/décroissant, ils sont permutés.
- Après chaque parcours complet du tableau, l'algorithme recommence l'opération.
- Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que le tableau est trié.

```
1   6 0 3 5 1 4 2   // On compare 6 et 0 : on inverse
2   0 6 3 5 1 4 2   // On compare 6 et 3 : on inverse
3   0 3 6 5 1 4 2   // On compare 6 et 5 : on inverse
4   0 3 5 6 1 4 2   // On compare 6 et 1 : on inverse
5   0 3 5 1 6 4 2   // On compare 6 et 4 : on inverse
6   0 3 5 1 4 6 2   // On compare 6 et 2 : on inverse
7   0 3 5 1 4 2 6   // Nous avons terminé notre premier passage
```

# Tri des tableaux

## Par bulle

### ■ Solution

```
Fonction tri(Entier tableau[]) retourne Entier[]
Entier i, tampon
Entier taille_tableau ← tableau->Taille
Booleen inversion
Faire
    inversion ← faux
    Pour i ← 0 A taille_tableau -2 Faire
        Si (tableau[i] > tableau[i + 1]) alors
            tampon ← tableau[i]
            tableau[i] ← tableau[i + 1]
            tableau[i + 1] ← tampon
            inversion ← vrai
        FinSi
    FinPour
    TantQue (inversion) // càd : inversion = vrai
    Retourner tableau
Fin fonction
```

*Premier parcours du tableau*

9	1	1	1
1	9	4	4
4	4	9	2
2	2	2	9

*Deuxieme parcours du tableau*

1	1	1	1
4	4	2	2
2	2	4	4
9	9	9	9

*Troisieme parcours du tableau*

1	1	1	1
2	2	2	2
4	4	4	4
9	9	9	9



# Tri des tableaux

## ■ Par permutation

- On parcourt le tableau jusqu'à ce que l'on trouve un élément plus petit que le précédent donc mal placé.
- On prend cet élément et on le range à sa bonne place dans le tableau puis on continue la lecture.
- On s'arrête à la fin du tableau.

Jeu d'essai	52	10	1	25	62	3	8	55	3	23
1ere boucle	10	52	1	25	62	3	8	55	3	23
2ieme boucle	1	10	52	25	62	3	8	55	3	23
3ieme boucle	1	10	25	52	62	3	8	55	3	23
4ieme boucle	1	3	10	25	52	62	8	55	3	23
5ieme boucle	1	3	8	10	25	52	62	55	3	23
6ieme boucle	1	3	8	10	25	52	55	62	3	23
7ieme boucle	1	3	3	8	10	25	52	55	62	23
8ieme boucle	1	3	3	8	10	23	25	52	55	62





# Tri des tableaux

## ■ Par permutation

### ■ Solution

```
Fonction tri(Entier tab[]) Retourne Entier[]
Entier i,j,k,save
Entier taille_tab ← tab→Taille
Pour i ← 1 A taille_tab - 1 Faire
  Si (tab[i] < tab[i-1]) Alors
    save ← tab[i]
    j ← 0
    TantQue (tab[j]<tab[i]) Faire
      j ← j+1
    FinTantQue
    Pour k ← i A j+1 PAS -1 Faire
      tab[k] ← tab[k-1]
    FinPour
    tab[j] ← save;
  FinSi
FinPour
Retourner tab
FinFonction
```

# Structures dynamiques



- **Une structure dynamique est une structure dont la taille n'est pas figée**
  - **la taille s'adapte au contenu**
  - C'est un enchainement de **structures composites**
  - Ex. Tableaux dynamiques, Listes, arbres, piles, ...

# Structures dynamiques

## ■ Structures 1/2

- Une structure est un ensemble de données
- Elle est définie comme un **type**

```
Structure TypeStructure  
    Type1 champs1,  
    Type2 champs2,  
    ...  
Fin Structure
```

```
TypeStructure struct1
```

```
Structure StructPoint  
    Entier x,  
    Entier y  
Fin Structure
```

```
StructPoint p
```

# Structures dynamiques

## ■ Structures 2/2

- Les champs dans une structure sont appelés membres
- L'accès aux membres de la structure se fait par  $\rightarrow$

```
TypeStructure struct1  
Struct1 $\rightarrow$ champs1  $\leftarrow$  valeur  
Type2 var2  $\leftarrow$  Struct1 $\rightarrow$ champs1
```

```
Structure TypeStructure  
    Type1 champs1,  
    Type2 champs2,  
    ...  
Fin Structure
```

```
StructPoint p  
 $p \rightarrow x$   $\leftarrow$  5  
Entier i =  $p \rightarrow x$ 
```

```
Structure StructPoint  
    Entier x,  
    Entier y  
Fin Structure
```

# Structures dynamiques

## Listes

- Une liste est obtenue par chainage de structures
- Pour chaîner une structure avec une autre structure,

- Les structures doivent posséder un champ pour le chainage
  - Dont le type est le type de la structure suivi de \*

```
Structure TypeStructure
    Type1 champs1,
    Type2 champs2,
    ...
    TypeStructure * suivant
Fin Structure
```

Pointeur

- La valeur du champ de chainage pour la première structure est l'adresse de la deuxième structure

```
TypeStructure struct1, struct2
struct1→suivant ← & struct2
```

& opérateur adresse

- La valeur du champ de chainage pour le dernier élément est **NULL**

```
struct2→suivant ← NULL
```

- La liste est désignée par sa tête qui contient l'adresse du premier élément

```
TypeStructure * liste1
liste1 ← & struct1
```

# Structures dynamiques

## Exemple de parcours de listes

```
TypeStructure* elt ← liste1
TantQue (elt ≠ NULL) Faire
    elt→champs1 ← valeur
    /***/
    elt ← elt→suivant
FinTantQue
```

Tel que

```
Structure TypeStructure
    Type1 champs1,
    Type2 champs2,
    ...
    TypeStructure * suivant
Fin Structure
```

```
TypeStructure* liste1
```

# Structures dynamiques

## Exercice

- Ecrire un algorithme d'une fonction inserer qui prend en argument une liste de StructPoint et un élément de type StructPoint et qui insère l'élément en tête de la liste

Fonction inserer(StructPoint\* liste, StructPoint p) Retourne StructPoint\*

Fin Fonction

- La structure StructPoint doit être adaptée pour être chaînée dans une liste

```
Structure StructPoint
    Entier x, y
    StructPoint* suivant
Fin Structure
```

# Structures dynamiques

## Exercice

- Ecrire un algorithme d'une fonction ajouter qui prend en argument une liste de StructPoint et un élément de type StructPoint et qui ajoute l'élément en queue de la liste

Fonction ajouter(StructPoint\* liste, StructPoint p) Retourne StructPoint \*

Fin Fonction



# Structures dynamiques

## Exercice

- Ecrire un algorithme d'une fonction ajouterN qui prend en argument une liste de StructPoint, une position entière et un élément de type StructPoint et qui ajoute l'élément à la position dans la liste
  - Si jamais la position dépasse le nombre d'éléments de la liste, l'élément est ajouté à la fin

Fonction ajouterN(StructPoint\* liste, StructPoint p, Entier position)  
Retourne StructPoint\*

Fin Fonction

