

---

# Cours d'algorithmique

# **Objectif et plan du cours**

---

- **Objectif:**
  - Apprendre les concepts de base de l'algorithmique et de la programmation
  - Etre capable de mettre en œuvre ces concepts pour analyser des problèmes simples et écrire les programmes correspondants
- **Plan:**
  - Généralités (matériel d'un ordinateur, systèmes d'exploitation, langages de programmation, ...)
  - Algorithmique (affectation, instructions conditionnelles, instructions itératives, fonctions, procédures, ...)

# Système Informatique?

- Techniques du traitement **automatique** de l'**information** au moyen des ordinateurs

Système informatique = ordinateur + périphériques

- Éléments d'un système informatique

Applications

(Word, Excel, Jeux, Maple, etc.)

Langages

(Java, C/C++, Fortran, etc.)

Système d'exploitation

(DOS, Windows, Unix, etc.)

Matériel

(PC, Macintosh, station SUN, etc.)

# Qu'est-ce qu'un programme d'ordinateur?

---

- Allumez un ordinateur, vous n'en tirerez rien!!
- Pour le faire marcher il faut lui fournir un programme  
    Ordinateur = matériel + programme(s)
- Un programme est une suite d'instructions d'ordinateur
- Une instruction est un **ordre** compris par l'ordinateur et qui lui fait exécuté une **action**, c-à-d une modification de son environnement

# Les catégories d'ordres

---

- les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on n'emploiera pas le terme d'ordre, mais plutôt celui d'instructions). Ces quatre familles d'instructions sont :
  - l'affectation de variables
  - la lecture / écriture
  - les tests
  - les boucles

## **Actions d'un ordinateur : Exemple**

---

- Attendre qu'un nombre soit tapé au clavier
- Sortir à l'écran le nombre entré
- Attendre qu'un nombre soit tapé au clavier
- Sortir à l'écran le nombre entré
- Additionner les deux nombres entrés
- Sortir à l'écran le résultat de l'addition



Ces lignes forment un programme d'ordinateur

# **Qu'est ce qu'un système d'exploitation?**

---

- Ensemble de programmes qui gèrent le matériel et contrôlent les applications
- **Gestion des périphériques** (affichage à l'écran, lecture du clavier, pilotage d'une imprimante, ...)
- **Gestion des utilisateurs et de leurs données** (comptes, partage des ressources, gestion des fichiers et répertoires, ...)
- **Interface avec l'utilisateur (textuelle ou graphique):** Interprétation des commandes
- **Contrôle des programmes** (découpage en tâches, partage du temps processeur, ...)

# Langages informatiques

---

- Un langage informatique est un outil permettant de donner des ordres (**instructions**) à la machine
  - A chaque instruction correspond une action du processeur
- Intérêt : écrire des **programmes** (suite consécutive d'instructions) destinés à effectuer une tâche donnée
  - Exemple: un programme de gestion de comptes bancaires
- Contrainte: être compréhensible par la machine

# Langage machine

---

- Langage **binnaire**: l'information est exprimée et manipulée sous forme d'une suite de bits
- Un **bit** (*binary digit*) = 0 ou 1 (2 états électriques)
  - Le code **ASCII** (*American Standard Code for Information Interchange*) donne les correspondances entre les caractères alphanumériques et leurs représentation binaire, Ex. A= 01000001, ?=00111111
- Les opérations logiques et arithmétiques de base (addition, multiplication, ... ) sont effectuées en binaire

# L'assembleur

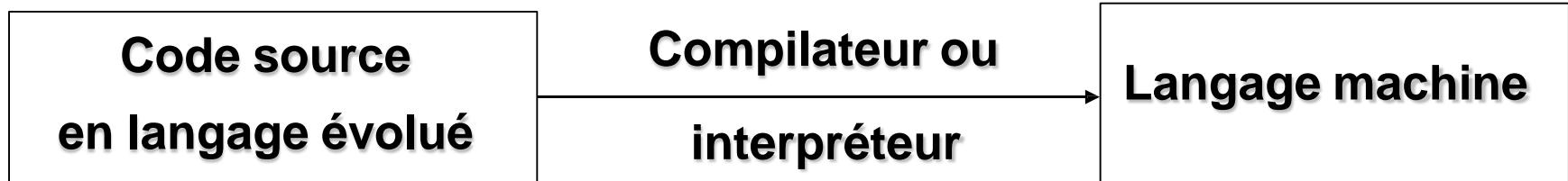
- Problème: le langage machine est difficile à comprendre par l'humain
- Idée: trouver un langage compréhensible par l'homme qui sera ensuite converti en langage machine
  - **Assembleur** (1er langage): exprimer les instructions élémentaires de façon symbolique
  - +: déjà plus accessible que le langage machine
  - -: dépend du type de la machine (n'est pas **portable**)
  - -: pas assez efficace pour développer des applications complexes

Apparition des langages évolués



# **Langages haut niveau**

- Intérêts multiples pour le haut niveau:
  - proche du langage humain «anglais» (compréhensible)
  - permet une plus grande portabilité (indépendant du matériel)
  - Manipulation de données et d'expressions complexes (réels, objets,  $a^*b/c$ , ...)
- Nécessité d'un traducteur (compilateur/interpréteur), exécution plus ou moins lente selon le traducteur



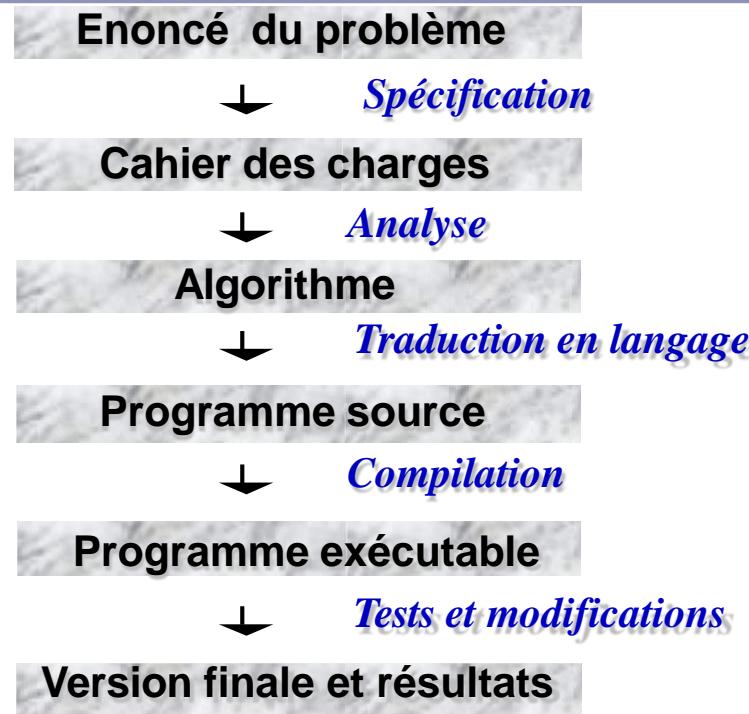
# Compilateur/interpréteur

- Compilateur: traduire le programme entier une fois pour toutes



- + plus rapide à l'exécution
- + sécurité du code source
- - il faut recompiler à chaque modification
- Interpréteur: traduire au fur et à mesure les instructions
- programme à chaque exécution
  - + exécution instantanée appréciable pour les débutants
  - - exécution lente par rapport à la compilation

# Etapes de réalisation d'un programme



La réalisation de programmes passe par l'écriture d'algorithmes  
D'où l'intérêt de  
l'**Algorithmique**

# Pourquoi apprendre l'algorithmique pour apprendre à programmer ?

- Un algorithme est une description complète et détaillée des actions à effectuer et de leur séquencement pour arriver à un résultat donné
  - Intérêt: séparation analyse/codage (pas de préoccupation de syntaxe) l'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.
  - Qualités: **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), ...
- Pour prendre une image, si un programme était une dissertation, l'algorithmique serait le plan, une fois mis de côté la rédaction et l'orthographe. Mieux faire d'abord le plan et rédiger ensuite que l'inverse...

# Représentation d'un algorithme

---

Historiquement, deux façons pour représenter un algorithme:

- **L'Organigramme:** représentation graphique avec des symboles (carrés, losanges, etc.)
  - offre une vue d'ensemble de l'algorithme
  - représentation quasiment abandonnée aujourd'hui
- **Le pseudo-code:** représentation textuelle avec une série de conventions ressemblant à un langage de programmation (sans les problèmes de syntaxe)
  - plus pratique pour écrire un algorithme
  - représentation largement utilisée

# Algorithmique

***Notions et instructions de base***

# Les catégories d'ordres

---

- les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on n'emploiera pas le terme d'ordre, mais plutôt celui d'instructions). Ces quatre familles d'instructions sont :
  - Les variables et leurs affectation
  - la lecture / écriture
  - les tests
  - les boucles

# Notion de variable

---

- Dans les langages de programmation une **variable** sert à stocker la valeur d'une donnée
- Une variable désigne en fait un emplacement mémoire dont le contenu peut changer au cours d'un programme (d'où le nom variable)
- Règle : Les variables doivent être **déclarées** avant d'être utilisées, elle doivent être caractérisées par :
  - un nom (**Identificateur**)
  - un **type** (entier, réel, caractère, chaîne de caractères, ...)

# Choix des identificateurs (1)

Le choix des noms de variables est soumis à quelques règles qui varient selon le langage, mais en général:

## Choix des identificateurs (2)

---

Conseil: pour la lisibilité du code choisir des noms significatifs qui décrivent les données manipulées

exemples: TotalVentes2006, Prix\_TTC, Prix\_HT

Remarque: en pseudo-code algorithme, on va respecter les règles citées, même si on est libre dans la syntaxe

# Types des variables

Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre, les types offerts par la plus part des langages sont:

- Type numérique (entier ou réel)
  - **Byte** (codé sur 1 octet): de 0 à 255
  - **Entier court** (codé sur 2 octets) : -32 768 à 32 767
  - **Entier long** (codé sur 4 ou 8 octets)
  - **Réel simple précision** (codé sur 4 octets)
  - **Réel double précision** (codé sur 8 octets)
- Type logique ou booléen: deux valeurs VRAI ou FAUX
- Type caractère: lettres majuscules, minuscules, chiffres, symboles, ...  
**exemples:** 'A', 'a', '1', '?', ...
- Type chaîne de caractère: toute suite de caractères,  
**exemples:** " Nom, Prénom", "code postale: 1000", ...

# Déclaration des variables

---

- Rappel: toute variable utilisée dans un programme doit avoir fait l'objet d'une déclaration préalable
- En pseudo-code, on va adopter la forme suivante pour la déclaration de variables

**Variables   liste d'identificateurs : type**

- Exemple:

**Variables   i, j,k : entier**

**x, y : réel**

**OK: booléen**

**ch1, ch2 : chaîne de caractères**

- Remarque: pour le type numérique on va se limiter aux entiers et réels sans considérer les sous types

# L'instruction d'affectation

- L'**affectation** consiste à attribuer une valeur à une variable (ça consiste en fait à remplir où à modifier le contenu d'une zone mémoire)
- En pseudo-code, l'affectation se note avec le signe  $\leftarrow$   
~~Var $\leftarrow$  e; attribue la valeur de e à la variable Var~~
  - e peut être une valeur, une autre variable ou une expression
  - Var et e doivent être **de même type** ou de types compatibles
  - l'affectation ne modifie que ce qui est à gauche de la flèche
- **Ex valides:**  $i \leftarrow 1$        $j \leftarrow i$        $k \leftarrow i+j$   
 $x \leftarrow 10.3$        $OK \leftarrow FAUX$        $ch1 \leftarrow "SMI"$   
 $ch2 \leftarrow ch1$        $x \leftarrow 4$        $x \leftarrow j$
- **non valides:**  $i \leftarrow 10.3$        $OK \leftarrow "SMI"$        $j \leftarrow x$

(voir la déclaration des variables dans le transparent précédent)

# Quelques remarques

---

- Beaucoup de langages de programmation (C/C++, Java, ...) utilisent le signe égal  $=$  pour l'affectation  $\leftarrow$ . Attention aux confusions:
  - l'affectation n'est pas commutative :  $A=B$  est différente de  $B=A$
  - l'affectation est différente d'une équation mathématique :
    - $A=A+1$  a un sens en langages de programmation
    - $A+1=2$  n'est pas possible en langages de programmation et n'est pas équivalente à  $A=1$
- Certains langages donnent des valeurs par défaut aux variables déclarées. Pour éviter tout problème il est préférable **d'initialiser les variables** déclarées

## **Exercices simples sur l'affectation (1)**

---

Donnez les valeurs des variables A, B et C après exécution des instructions suivantes ?

**Variables A, B, C: Entier**

**Début**

A ← 3

B ← 7

A ← B

B ← A+5

C ← A + B

C ← B – A

**Fin**

## **Exercices simples sur l'affectation (2)**

---

Donnez les valeurs des variables A et B après exécution des instructions suivantes ?

**Variables** A, B : Entier

**Début**

A  $\leftarrow$  1

B  $\leftarrow$  2

A  $\leftarrow$  B

B  $\leftarrow$  A

**Fin**

Les deux dernières instructions permettent-elles d'échanger les valeurs de A et B ?

# Expressions et opérateurs

- Une **expression** peut être une valeur, une variable ou une opération constituée de variables reliées par des **opérateurs**      **exemples:** 1, b, a<sup>2</sup>, a+ 3\*b-c, ...
- L'évaluation de l'expression fournit une valeur unique qui est le résultat de l'opération
- Les **opérateurs** dépendent du type de l'opération, ils peuvent être :
  - des opérateurs arithmétiques: +, -, \*, /, % (modulo), ^ (puissance)
  - des opérateurs logiques: NON, OU, ET
  - des opérateurs relationnels: =, ≠ , <, >, <=, >=
  - des opérateurs sur les chaînes: & (concaténation)
- Une expression est évaluée de gauche à droite mais en tenant compte de **priorités**

# Priorité des opérateurs

- Pour les opérateurs arithmétiques donnés ci-dessus, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :
  - $\wedge$  : (élévation à la puissance)
  - $*$ ,  $/$  (multiplication, division)
  - $\%$  (modulo)
  - $+$ ,  $-$  (addition, soustraction)
- En cas de besoin (ou de doute), on utilise les parenthèses pour indiquer les opérations à effectuer en priorité

exemple:  
**exemple:**  $2 + 3 * 7$  vaut  $23$

# Les instructions d'entrées-sorties: lecture et écriture (1)

- Les instructions de lecture et d'écriture permettent à la machine de communiquer avec l'utilisateur
- La **lecture** permet d'entrer des donnés à partir du clavier
  - En pseudo-code, on note: **lire (var)**  
la machine met la valeur entrée au clavier dans la zone mémoire nommée var
  - Remarque: Le programme s'arrête lorsqu'il rencontre une instruction Lire et ne se poursuit qu'après la frappe d'une valeur au clavier et de la touche Entrée

# Les instructions d'entrées-sorties: lecture et écriture (2)

- L'**écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
  - En pseudo-code, on note: **écrire (var)**  
la machine affiche le contenu de la zone mémoire var
  - Conseil: Avant de lire une variable, il est fortement conseillé d'écrire des messages à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper

# **Exemple** (lecture et écriture)

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui calcule et affiche le double de ce nombre

## **Algorithme Calcul\_double**

**variables** A, B : entier

**Début**

**écrire**("entrer la valeur de A ")

**lire**(A)

B  $\leftarrow$  2\*A

**écrire**("le double de ", A, "est :", B)

**Fin**

# **Exercice** (lecture et écriture)

Ecrire un algorithme qui vous demande de saisir votre nom puis votre prénom et qui affiche ensuite votre nom complet

## **Algorithme AffichageNomComplet**

**variables** Nom, Prenom, Nom\_Complet : **chaîne de caractères**

**Début**

**écrire**("entrez votre nom")

**lire**(Nom)

**écrire**("entrez votre prénom")

**lire**(Prenom)

    Nom\_Complet ← Nom & Prenom

**écrire**("Votre nom complet est : ", Nom\_Complet)

**Fin**

## **Méthode de construction d'un algorithme simple (1/4)**

---

Exemple :

Écrire un algorithme qui consiste à calculer l'air S d'un cercle selon la formule  $S = \pi * R^2$

Rappel :  $\pi = 3.14159$  et R le rayon du cercle

## Méthode de construction d'un algorithme simple (2/4)

---

Méthodologie à suivre :

- **Constantes** : Pi = 3.14159
- **Variables** : Rayon, Surface
- **Types** : Rayon, Surface : réel
- **Expressions et affectation** : Surface := Pi \* (Rayon)<sup>2</sup>
- **Structures conditionnelles et les boucles** : -----
- **Opérations d'entrée-sortie** : Lire (Rayon),  
Écrire (Surface)

# Méthode de construction d'un algorithme simple (3/4)

Algorithme

**Calcul\_Aire**

**Constantes**

Pi = 3,14159

**Variables**

Rayon, Surface : **réels**

**Début**

**lire** (Rayon)

Surface := Pi \* (Rayon)<sup>2</sup>

**écrire** (Surface)

**Fin**



# Méthode de construction d'un algorithme simple (3/4)

## Programme C

```
#include <stdio.h>
#include <math.h>
Main ( )
{
    Float Pi = 3.14159;
    Float rayon, surface;
    Scanf (« °/°f », &rayon);
    surface = pi*pow (rayon,2);
    Printif (« °/°f\n »surface, )
    Return 0;
}
```

# **Algorithmique**

**Les structures  
Conditionnelles et les  
boucles**

# Besoin a des concepts de ruptures de séquence

## Algorithme

**Calcul\_Aire**

**Constantes**

Pi = 3,14159

**Variables**

Rayon, Surface : **réels**

**Début**

**lire** (Rayon)

Surface := Pi \* (Rayon)<sup>2</sup>

**écrire** (Surface)

**Fin**

- Rare les algorithme qui peuvent se décrire uniquement par un enchaînement séquentiel d'opération élémentaire



- On a besoin a des concept de rupture de séquence comme les test et les boucles

**Ex:**

- ✓ un algorithme qui résout une équation de deuxième degré
- ✓ un algorithme qui calcule une série numérique

# **Les structures conditionnelles et les boucles**

---

- **Les tests simples** : permet de réaliser un choix parmi deux possibilités (Ex : Booléenne : vrais ou faux)
- **Les instructions conditionnelles** : c'est un concept de tests multiples, permet de comparer un objet à une série de valeurs, et exécuter si la condition est vérifier (Ex : recherche des nombres premier dans une ensemble)
- **Les itérations** : consiste a exécuté un bloc d'instructions un certain nombre de fois (Ex : calcul d'une suite numérique)
- **Les boucles conditionnelles** : consiste a exécuté un bloc d'instructions un certain nombre de fois si la condition est vérifier (Ex : On veut afficher le 100 premiers nombres :. Tant que i est plus petit que 100, afficher la valeur de i).

# Tests: instructions conditionnelles (1)

- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée
  - On utilisera la forme suivante: **Si** condition **alors**  
instruction ou suite d'instructions1  
**Sinon**  
instruction ou suite d'instructions2  
**Finsi**
  - la condition ne peut être que vraie ou fausse
  - si la condition est vraie, se sont les instructions1 qui seront exécutées
  - si la condition est fausse, se sont les instructions2 qui seront exécutées
  - la condition peut être une condition simple ou une condition composée de plusieurs conditions

## **Tests: instructions conditionnelles (2)**

---

- La partie Sinon n'est pas obligatoire, quand elle n'existe pas et que la condition est fausse, aucun traitement n'est réalisé
  - On utilisera dans ce cas la forme simplifiée suivante:

**Si** condition **alors**

instruction ou suite d'instructions1

**Finsi**

# **Exemple (Si...Alors...Sinon)**

---

**Algorithme AffichageValeurAbsolue (version1)**

**Variable x : réel**

**Début**

**Ecrire " Entrez un réel : "**

**Lire (x)**

**Si x < 0 alors**

**Ecrire ("la valeur absolue de ", x, "est:",-x)**

**Sinon**

**Ecrire ("la valeur absolue de ", x, "est:",x)**

**Finsi**

**Fin**

# **Exemple (Si...Alors)**

---

**Algorithme AffichageValeurAbsolue (version2)**

**Variable** x,y : réel

**Début**

**Ecrire** " Entrez un réel : "

**Lire** (x)

y $\leftarrow$  x

**Si** x < 0 **alors**

y  $\leftarrow$  -x

**Finsi**

**Ecrire** ("la valeur absolue de ", x, "est:",y)

**Fin**

# **Exercice (tests)**

---

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui teste et affiche s'il est divisible par 3

**Algorithme Divsible\_par3**

**Variable n : entier**

**Début**

**Ecrire " Entrez un entier : "**

**Lire (n)**

**Si (n%3=0) alors**

**Ecrire (n," est divisible par 3")**

**Sinon**

**Ecrire (n," n'est pas divisible par 3")**

**Finsi**

**Fin**

# Conditions composées

---

- Une condition composée est une condition formée de plusieurs conditions simples reliées par des opérateurs logiques:  
ET, OU, OU exclusif (XOR) et NON
- Exemples :
  - x compris entre 2 et 6 :  $(x > 2)$  ET  $(x < 6)$
  - n divisible par 3 ou par 2 :  $(n \% 3 = 0)$  OU  $(n \% 2 = 0)$
  - deux valeurs et deux seulement sont identiques parmi a, b et c :  
 $(a=b)$  XOR  $(a=c)$  XOR  $(b=c)$
- L'évaluation d'une condition composée se fait selon des règles présentées généralement dans ce qu'on appelle tables de vérité

# Tables de vérité

C1	C2	C1 ET C2	C1	C2	C1 OU C2
VRAI	VRAI	<b>VRAI</b>	VRAI	VRAI	<b>VRAI</b>
VRAI	FAUX	<b>FAUX</b>	VRAI	FAUX	<b>VRAI</b>
FAUX	VRAI	<b>FAUX</b>	FAUX	VRAI	<b>VRAI</b>
FAUX	FAUX	<b>FAUX</b>	FAUX	FAUX	<b>FAUX</b>

C1	NON C1
VRAI	<b>FAUX</b>
FAUX	<b>VRAI</b>

# Tests imbriqués

- Les tests peuvent avoir un degré quelconque d'imbrications

**Si condition1 alors**

**Si condition2 alors**

instructionsA

**Sinon**

instructionsB

**Finsi**

**Sinon**

**Si condition3 alors**

instructionsC

**Finsi**

**Finsi**

## Tests imbriqués: exemple (version 1)

Variable n : entier

**Début**

Ecrire ("entrez un nombre : ")

Lire (n)

**Si** n < 0 **alors**

Ecrire ("Ce nombre est négatif")

**Sinon**

**Si** n = 0 **alors**

Ecrire ("Ce nombre est nul")

**Sinon**

Ecrire ("Ce nombre est positif")

**Finsi**

**Finsi**

**Fin**

## Tests imbriqués: exemple (version 2)

Variable n : entier

**Début**

Ecrire ("entrez un nombre : ")

Lire (n)

**Si** n < 0 **alors**      Ecrire ("Ce nombre est négatif")

**Finsi**

**Si** n = 0 **alors**      Ecrire ("Ce nombre est nul")

**Finsi**

**Si** n > 0 **alors**      Ecrire ("Ce nombre est positif")

**Finsi**

**Fin**

**Remarque :** dans la version 2 on fait trois tests systématiquement alors que dans la version 1, si le nombre est négatif on ne fait qu'un seul test

**Conseil :** utiliser les tests imbriqués pour limiter le nombre de tests et placer d'abord les conditions les plus probables (minimiser la complexité)

## Tests imbriqués: exercice

---

Le prix de photocopies dans une reprographie varie selon le nombre demandé: 0,5 DH la copie pour un nombre de copies inférieur à 10, 0,4DH pour un nombre compris entre 10 et 20 et 0,3DH au-delà.

Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées, qui calcule et affiche le prix à payer

# Tests imbriqués: corrigé de l'exercice

---

**Variables** copies : entier  
                  prix : réel

**Début**

Ecrire ("Nombre de photocopies : ")

Lire (copies)

**Si** copies < 10 **Alors**

    prix ← copies\*0.5

**Sinon Si** copies < 20

        prix ← copies\*0.4

**Sinon**

        prix ← copies\*0.3

**Finsi**

**Finsi**

Ecrire ("Le prix à payer est : ", prix)

**Fin**

## Tests imbriqués: Exercice 2

---

Écrire l'algorithme du traitement qui calcule le discriminant DELTA d'un trinôme du second degré  $AX^2 + BX + C$  et qui, en fonction de son signe, calcule la ou les racines réelles du trinôme ou affiche, si besoin est qu'il n'y a pas de racine réelle.

Les trois coefficients A, B et C seront saisis au clavier avant traitement.

# Tests imbriqués: corrigé de l'exercice 2

## Variables

A, B, C, Delta, X1, X2 : réels

## Début

Lire (A, B, C)

Delta  $\leftarrow B^2 - 4AC$

## Si (Delta < 0) Alors

Ecrire (« le trinôme n'a pas de racine réelle »)

## Sinon

### Si (Delta > 0 Alors

X1  $\leftarrow (-B + \sqrt{\Delta}) / 2A$

X2  $\leftarrow (-B - \sqrt{\Delta}) / 2A$

Ecrire (« le trinôme possède deux racines réelles : », X1, X2)

### Sinon

X1  $\leftarrow (-B) / 2A$

Ecrire (« le trinôme possède une racine réelle : », X1)

## Finsi

## Finsi

## Fin

# **Algorithmique**

**Les boucles**

# Les types de boucle

---

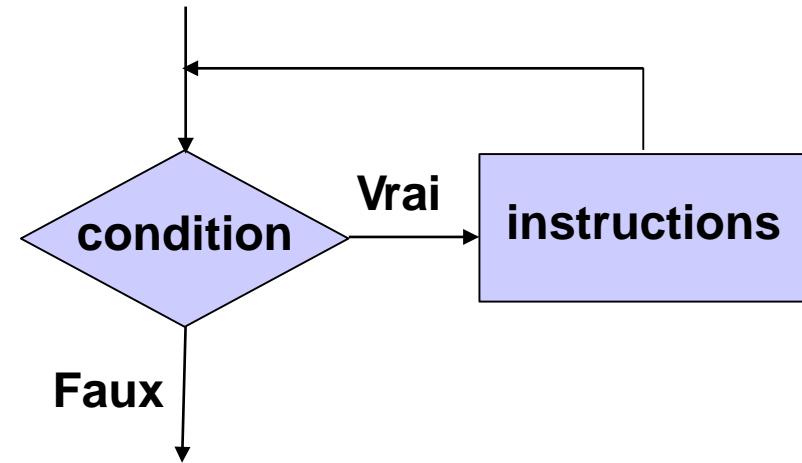
- On distingue 2 types de boucles:
  - **Les boucles à compteur ou définie**
    - On sait à l'avance combien de fois la boucle devra tourner et une variable (le compteur ) compte les répétitions
      - Choisir 10 nombres au hasard. On fera dix fois l'opération choisir un nombre au hasard.
      - Ex : **la boucle Pour**
  - **Les boucles à événement ou indéfinie**
    - On ne sait pas à l'avance le nombre de fois que la boucle sera exécutée.
      - Ça peut dépendre du nombre de données à traiter.
      - Ça peut dépendre du nombre d'essais que l'usager a effectués.
      - Ex : **la boucle Tantque** et **la boucle jusqu'à**

# Les boucles Tant que

**TantQue** (condition)

instructions

**FinTantQue**



# Boucle Tant que : exemple simple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

Variables som, i : entier

**Debut**

i  $\leftarrow$  0

som  $\leftarrow$  0

**TantQue** (som  $\leq$  100)

i  $\leftarrow$  i+1

som  $\leftarrow$  som+i

**FinTantQue**

Ecrire (" La valeur cherchée est N= ", i)

**Fin**

# Boucle Tant que : exemple

**Algorithme Plus-Grand-Element:** Retourne la plus grande valeur d'une liste

**Entrée:**  $n$  entiers  $S_1, \dots, S_n$

**Sortie:**  $grand$  contenant le plus grand élément

**Algo** plus-grand ( $S, n$ )

$grand := S_1;$

$i := 2;$

**Tant que**  $i \leq n$  **Faire**

**Début**

**Si**  $S_i > grand$  **alors** // une plus grande valeur a été trouvée

$grand := S_i;$

$i := i + 1;$

**Fin**

**ecrire** ( $grand$ )

**Fin** plus-grand;

**Trace de l'algorithme:**

$n=4; S_1 = -2; S_2 = 6; S_3 = 5; S_4 = 6$

$grand = -2$

$i = 2$

$grand = 6$

$i = 3$

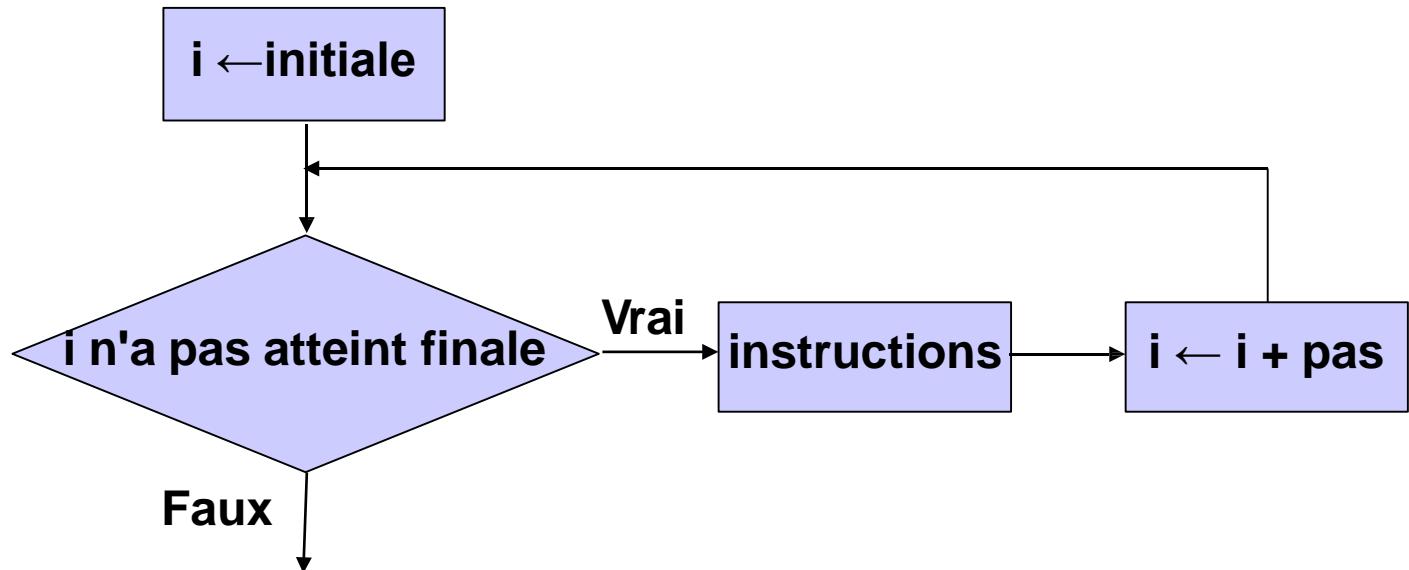
$i = 4$

$i = 5$

# Les boucles Pour

**Pour** compteur **allant de** initiale **à** finale par **pas** valeur du pas  
instructions

**FinPour**



# Les boucles Pour

---

Remarques :

- **Compteur** est une variable de type entier (ou caractère). Elle doit être déclarée
- **Pas** est un entier qui peut être positif ou négatif. **Pas** peut ne pas être mentionné, car par défaut sa valeur est égal à 1. Dans ce cas, le nombre d'itérations est égal à finale - initiale+ 1
- **Initiale et finale** peuvent être des valeurs, des variables définies avant le début de la boucle ou des expressions de même type que compteur

# Déroulement des boucles Pour

---

- 1) La valeur initiale est affectée à la variable compteur
- 2) On compare la valeur du compteur et la valeur de finale :
  - a) Si la valeur du **compteur** est  $>$  à la **valeur finale** dans le cas d'un **pas** positif (ou si compteur est  $<$  à finale pour un **pas** négatif), on sort de la boucle et on continue avec l'instruction qui suit FinPour
  - b) Si **compteur** est  $\leq$  à **finale** dans le cas d'un **pas** positif (ou si compteur est  $\geq$  à finale pour un **pas** négatif), instructions seront exécutées
    - i. Ensuite, la valeur de **compteur** est incrémentée de la valeur du **pas** si **pas** est positif (ou décrémenté si **pas** est négatif)
    - ii. On recommence l'étape 2 : La comparaison entre compteur et finale est de nouveau effectuée, et ainsi de suite ...

# Boucle Pour : exemple

---

**Algorithme Plus-Grand-Element:** Réécriture de l'algorithme précédent mais avec une boucle ``Pour''

**Entrée:**  $n$  entiers  $S_1, \dots, S_n$

**Sortie:**  $grand$  contenant le plus grand élément

**Algo** *plus-grand* ( $S, n$ )

$grand := S_1;$

**Pour**  $i = 1$  à  $n$  **Faire**

**Si**  $S_i > grand$  **alors** // une plus grande valeur a été trouvée

$grand := S_i;$

**écrire** ( $grand$ )

**Fin** *plus-grand*;

# Boucle Pour : remarque

---

- Il faut éviter de modifier la valeur du compteur (et de finale) à l'intérieur de la boucle. En effet, une telle action :
  - perturbe le nombre d'itérations prévu par la boucle Pour
  - rend difficile la lecture de l'algorithme
  - présente le risque d'aboutir à une boucle infinie

Exemple : **Pour i allant de 1 à 5**

$i \leftarrow i - 1$

**écrire(" i = ", i)**

**Finpour**

# Lien entre Pour et TantQue

La boucle **Pour** est un cas particulier de **Tant Que** (cas où le nombre d'itérations est connu et fixé) . Tout ce qu'on peut écrire avec Pour peut être remplacé avec TantQue (la réciproque est fausse)

**Pour** compteur **allant de** initiale **à** finale par **pas** valeur du pas  
instructions

## **FinPour**

peut être remplacé par :  
(cas d'un pas positif)

compteur  $\leftarrow$  initiale  
**TantQue** compteur  $\leq$  finale  
instructions  
compteur  $\leftarrow$  compteur+pas

## **FinTantQue**

## Lien entre Pour et TantQue: exemple

---

Calcul de **x** à la puissance **n**  
avec la boucle **Pour** et la  
boucle **TantQue**

**x** : un réel non nul

**n** : entier positif ou nul

# Solution avec la boucle Pour

---

Variables x, puiss : réel  
n, i : entier

**Debut**

Ecrire (" Entrez respectivement les valeurs de x et n ")

Lire (x, n)

puiss  $\leftarrow$  1

**Pour i allant de 1 à n**

    puiss  $\leftarrow$  puiss\*x

**FinPour**

Ecrire (x, " à la puissance ", n, " est égal à ", puiss)

**Fin**

# Solution avec la boucle Tant Que

---

Variables x, puiss : réel  
n, i : entier

## Debut

Ecrire (" Entrez respectivement les valeurs de x et n ")

Lire (x, n)

puiss  $\leftarrow$  1, i  $\leftarrow$  1

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

    puiss  $\leftarrow$  puiss\*x

    i  $\leftarrow$  i+1

**FinTantQue**

Ecrire (x, " à la puissance ", n, " est égal à ", puiss)

**Fin**

# Algorithme de la fonction factorielle : Exemple

---

- Écrire deux algorithmes qui calculent pour un entier positif donné **n** la valeur **n!**, un de ces algorithmes doit utiliser la boucle **Pour** et l'autre la boucle **Tanque**

Entrée : n de type naturel

Sortie : factoriel (n) =  $1*2*3*.....*(n-1)*n$

# Algorithme de la fonction factorielle

## Algorithme / tantque

Calcul factorielle 1

Variables

i, f, n : Naturel

Début

i  $\leftarrow$  1

f  $\leftarrow$  1

tant que (i < n)

i  $\leftarrow$  i+1

f  $\leftarrow$  f \* i

Fin de tant que

écrire (f)

Fin

## Algorithme / Pour

Calcul factorielle 2

Variables

i, f, n : Naturel

Début

f  $\leftarrow$  1

pour i variant de 2 à n

f  $\leftarrow$  f \* i

Fin pour

écrire (f)

Fin

## Algorithme de la recherche des nombres premiers : Exemple

- Problème: Écrire l'algorithme `estPremier`, qui a partir d'un entier strictement positif donné, retourne le résultat booléen `VRAI` ou `FAUX` selon le nombre est premier ou non.
- Procédure: pour déterminer si un entier  $m$  est un nombre premier. Il suffit de tester si  $m$  est divisible par un entier entre 2 et  $m/2$ 
  - Ex : 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47. (*listes des nombres premier <=50*)

# La boucle tantque en langage de programmation

Langage Pascal

**While** condition **Do**

Begin

    Bloc d'instructions;

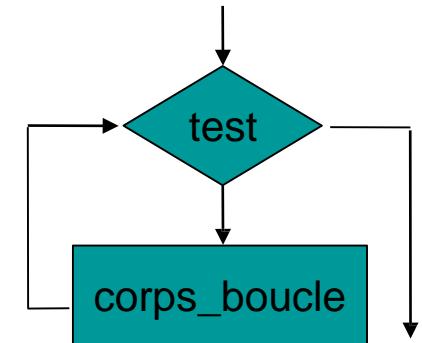
End;

Langage C et Java

**while** (condition) {

    Bloc d'instructions;

}



# La boucle Pour en langage de programmation

Langage Pascal

**For** variable := valeur initiale **To** valeur finale **Do**

Begin

Bloc d'instructions

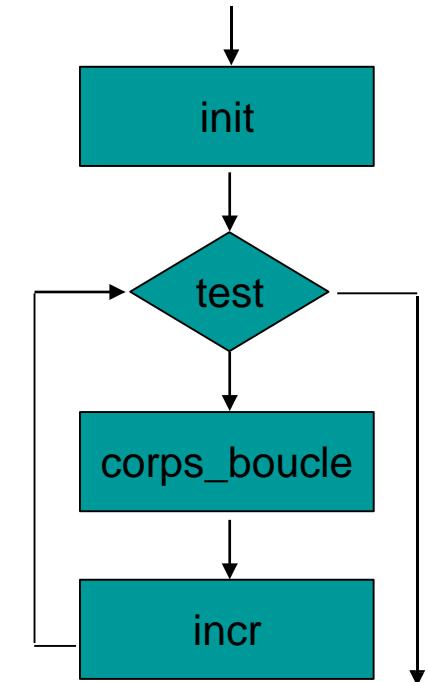
End;

Langage C et Java

**for** (*i*=valeur initiale; *i*<= valeur finale; *i*++) {

Bloc d'instructions;

}



# Déetecter l'erreur dans les deux essaie tant que et pour

Algorithme

**Essai de tant que**

**Variables**

n : entier

**Début**

n ← 15

**tant que** (n<>0)

    écrire (n)

    n ← n - 2

**Fin de tant que**

**Fin**

Algorithme

**Essai pour**

**Variables**

K, N : entier

**Début**

n ← 200

**pour K variant de 1 à n**

    écrire (K)

    K ← n – 100

**Fin pour**

**Fin**

# Boucles imbriquées

- Les instructions d'une boucle peuvent être des instructions itératives. Dans ce cas, on aboutit à des **boucles imbriquées**
- Exemple:

Pour i allant de 1 à 5

    Pour j allant de 1 à i

        écrire("O")

    FinPour

    écrire("X")

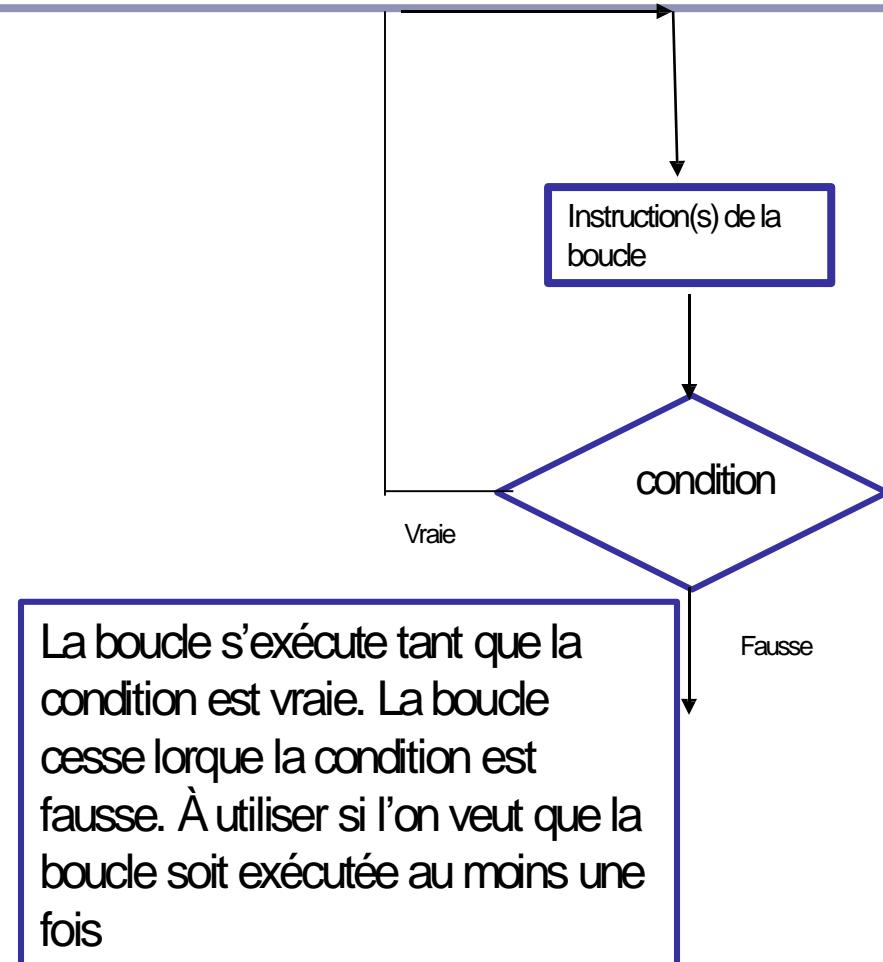
FinPour

## Exécution

# La boucle Faire...Tant Que

- Faire  
    Instruction(s)  
    Tant que (condition)
- do  
    Instruction;  
    while (condition)

```
do
{
    f :=n;
    n--;
} while (n>1);
```

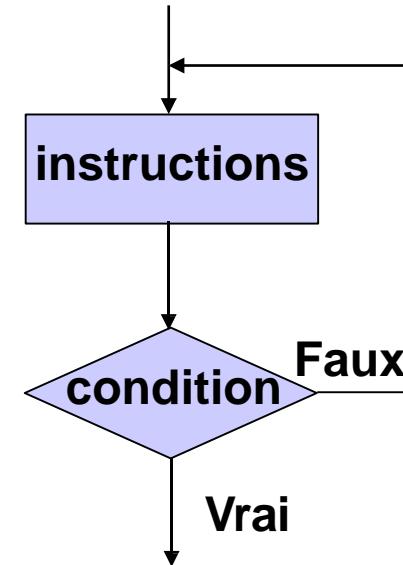


# Les boucles Répéter ... jusqu'à ...

Répéter

instructions

Jusqu'à      condition



- Condition est évaluée après chaque itération
- les instructions entre *Répéter* et *jusqu'à* sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que condition soit vrai (tant qu'elle est fausse)

## Boucle Répéter jusqu'à : exemple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (**version avec répéter jusqu'à**)

Variables som, i : entier

**Debut**

    som  $\leftarrow$  0

    i  $\leftarrow$  0

**Répéter**

        i  $\leftarrow$  i+1

        som  $\leftarrow$  som+i

**Jusqu'à** ( som > 100)

Ecrire (" La valeur cherchée est N= ", i)

**Fin**

## Choix d'un type de boucle

---

- Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser *la boucle Pour*
- S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des *boucles TantQue ou répéter jusqu'à*
- Pour le choix entre *TantQue* et *jusqu'à* :
  - Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera *TantQue*
  - Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera *répéter jusqu'à ou faire tanque*

# Algorithme de la racine carrée : Exercice

---

- Problème: Écrire l'algorithme qui calcul la racine carrée d'un nombre sans avoir recours a la fonction mathématique Racine Carrée prédéfinie.
- Procédure: si n est le nombre dont on souhaite extraire la racine carrée. On construit une suite de nombres  $X_i$  dont le premier vaut 1 et dont le terme générale a pour expression :

$$X_i = (n/x_{i-1} + x_{i-1}) / 2$$

Rq : Cette suite converge systématiquement vers racarrée (n)

# Algorithme de la racine carrée

---

## Algorithme Racine Carrée (RC)

### Variables

n, x : réels

i, max : entier

### Début

Lire (n)

Lire (max)

x  $\leftarrow$  1

pour i variant de 1 à max

    écrire (n)

    x  $\leftarrow$  ((n/x) + x) / 2

Fin de pour

Fin

# Transmission des paramètres

Il existe deux modes de transmission de paramètres dans les langages de programmation :

- **La transmission par valeur** : les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification
- **La transmission par adresse (ou par référence)** : les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution de la procédure
  - **Remarque** : le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse
  - En pseudo-code, on va préciser explicitement le mode de transmission dans la déclaration de la procédure

# Transmission des paramètres : exemples

**Procédure** incrementer1 (**x** : entier , **y** : entier )

    x  $\leftarrow$  x+1

    y  $\leftarrow$  y+1

**FinProcédure**

**Algorithme Test\_incrementer1**

variables    n, m : entier

**Début**

    n  $\leftarrow$  3

    m  $\leftarrow$  3

    incrementer1(n, m)

    écrire (" n= ", n, " et m= ", m)

**Fin**

**résultat:**

**Remarque :** l'instruction  $x \leftarrow x+1$  n'a pas de sens avec un passage par valeur

## **Transmission par valeur, par adresse : exemples**

---

Procédure qui calcule la somme et le produit de deux entiers :

**Procédure** SommeProduit (*x,y: entier, som, prod : entier*)

```
som ← x+y  
prod ← x*y
```

**FinProcédure**

Procédure qui échange le contenu de deux variables :

**Procédure** Echange (*x : réel , y : réel*)

variables z : réel

```
z ← x  
x ← y  
y ← z
```

**FinProcédure**

# Variables locales et globales (1)

---

- On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie")
- Une **variable locale** n'est connue qu'à l'intérieur du module où elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution
- Une **variable globale** est connue par l'ensemble des modules et le programme principal. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme

# Variables locales et globales (2)

- La manière de distinguer la déclaration des variables locales et globales diffère selon le langage
  - En général, les variables déclarées à l'intérieur d'une fonction ou procédure sont considérées comme variables locales
- En pseudo-code, on va adopter cette règle pour les variables locales et on déclarera les variables globales dans le programme principale
- **Conseil :** Il faut utiliser autant que possible des variables locales plutôt que des variables globales. Ceci permet d'économiser la mémoire et d'assurer l'indépendance de la procédure ou de la fonction

# Algorithmique

*Les tableaux*

# Exemple introductif

- Supposons qu'on veut conserver les notes d'une classe de 30 étudiants pour extraire quelques informations. Par exemple : calcul du nombre d'étudiants ayant une note supérieure à 10
- Le seul moyen dont nous disposons actuellement consiste à déclarer 30 variables, par exemple **N1**, ..., **N30**. Après 30 instructions lire, on doit écrire 30 instructions Si pour faire le calcul

**nbre**  $\leftarrow$  0

**Si (N1 >10) alors nbre  $\leftarrow$ nbre+1 FinSi**

....

**Si (N30>10) alors nbre  $\leftarrow$ nbre+1 FinSi**

c'est lourd à écrire

- Heureusement, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans **une seule structure de donnée** appelée **tableau**

# Tableaux

- Un **tableau** est un ensemble d'éléments de même type désignés par un identificateur unique
- Une variable entière nommée **indice** permet d'indiquer la position d'un élément donné au sein du tableau et de déterminer sa valeur
- La **déclaration** d'un tableau s'effectue en précisant le **type** de ses éléments et sa **dimension** (le nombre de ses éléments)
  - En pseudo code :  
**variable tableau identificateur[dimension] : type**
  - Exemple :  
**variable tableau notes[30] : réel**
- On peut définir des tableaux de tous types : tableaux d'entiers, de réels, de caractères, de booléens, de chaînes de caractères, ...

## Tableaux : remarques

---

- L'accès à un élément du tableau se fait au moyen de l'indice. Par exemple, **notes[i]** donne la valeur de l'élément i du tableau notes
- Selon les langages, le premier indice du tableau est soit 0, soit 1. Le plus souvent c'est 0 (c'est ce qu'on va adopter en pseudo-code). Dans ce cas, **notes[i]** désigne l'élément  $i+1$  du tableau notes
- Il est possible de déclarer un tableau sans préciser au départ sa dimension. Cette précision est faite ultérieurement
  - Par exemple, quand on déclare un tableau comme paramètre d'une procédure, on peut ne préciser sa dimension qu'au moment de l'appel
  - En tous cas, un tableau est inutilisable tant qu'on n'a pas précisé le nombre de ses éléments
- Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles

# Tableaux : exemples (1)

- Pour le calcul du nombre d'étudiants ayant une note supérieure à 10 avec les tableaux, on peut écrire :

Variables      i ,nbre : entier

**tableau** notes[30] : réel

**Début**

nbre  $\leftarrow$  0

**Pour** i allant de 0 à 29

**Si** (notes[i] >10) alors

        nbre  $\leftarrow$  nbre+1

**FinSi**

**FinPour**

écrire ("le nombre de notes supérieures à 10 est : ", nbre)

**Fin**

# Tableaux : saisie et affichage

- Procédures qui permettent de saisir et d'afficher les éléments d'un tableau :

**Procédure** SaisieTab(*n* : entier par valeur, **tableau** *T* : réel par référence )  
variable *i*: entier

**Pour** *i* allant de 0 à *n*-1  
    écrire ("Saisie de l'élément ", *i* + 1)  
    lire (*T[i]* )

**FinPour**

**Fin Procédure**

**Procédure** AfficheTab(*n* : entier par valeur, **tableau** *T* : réel par valeur )  
variable *i*: entier

**Pour** *i* allant de 0 à *n*-1  
    écrire ("T[",*i*, "] =", *T[i]*)

**FinPour**

**Fin Procédure**

# Tableaux : exemples d'appel

- Algorithme principale où on fait l'appel des procédures SaisieTab et AfficheTab :

## Algorithme Tableaux

variable p : entier

tableau A[10] : réel

Début

p  $\leftarrow$  10

SaisieTab(p, A)

AfficheTab(p, A)

Fin

# Tableaux : fonction longueur

La plus part des langages offrent une fonction **longueur** qui donne la dimension du tableau. Les procédures Saisie et Affiche peuvent être réécrites comme suit :

**Procédure** SaisieTab(**tableau** T : réel par référence )

variable i: entier

**Pour** i allant de 0 à **longueur(T)**-1

écrire ("Saisie de l'élément ", i + 1)

lire (T[i] )

**FinPour**

**Fin Procédure**

**Procédure** AfficheTab(**tableau** T : réel par valeur )

variable i: entier

**Pour** i allant de 0 à **longueur(T)**-1

écrire ("T[",i, "] =", T[i])

**FinPour**

**Fin Procédure**

# Tableaux à deux dimensions

- Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par **deux indices**. Ceci est utile par exemple pour représenter des matrices
- En pseudo code, un tableau à deux dimensions se **déclare** ainsi :  
variable **tableau** identificateur**[dimension1] [dimension2]** : type
  - Exemple : une matrice A de 3 lignes et 4 colonnes dont les éléments sont réels  
variable **tableau A[3][4]** : réel
  - **A[i][j]** permet d'accéder à l'élément de la matrice qui se trouve à l'intersection de la ligne i et de la colonne j

# Exemples : lecture d'une matrice

- Procédure qui permet de saisir les éléments d'une matrice :

**Procédure** SaisieMatrice(n : entier par valeur, m : entier par valeur,  
tableau A : réel par référence)

**Début**

variables i,j : entier

**Pour** i allant de 0 à n-1

    écrire ("saisie de la ligne ", i + 1)

**Pour** j allant de 0 à m-1

        écrire ("Entrez l'élément de la ligne ", i + 1, " et de la colonne ", j+1)

        lire (A[i][j])

**FinPour**

**FinPour**

**Fin Procédure**

## Exemples : affichage d'une matrice

- Procédure qui permet d'afficher les éléments d'une matrice :

**Procédure** AfficheMatrice(n : entier par valeur, m : entier par valeur,  
**tableau** A : réel par valeur)

**Début**

variables i,j : entier

**Pour** i allant de 0 à n-1

**Pour** j allant de 0 à m-1

écrire ("A[",i, "] [",j,"]=", A[i][j])

**FinPour**

**FinPour**

**Fin Procédure**

## **Exemples : somme de deux matrices**

---

- Procédure qui calcule la somme de deux matrices :

**Procédure** SommeMatrices(*n, m : entier par valeur,*

**tableau A, B : réel par valeur , tableau C : réel par référence )**

**Début**

variables *i,j : entier*

**Pour** *i* allant de 0 à *n-1*

**Pour** *j* allant de 0 à *m-1*

*C[i][j] ← A[i][j]+B[i][j]*

**FinPour**

**FinPour**

**Fin Procédure**

## **Appel des procédures définies sur les matrices**

---

Exemple d'algorithme principale où on fait l'appel des procédures définies précédemment pour la saisie, l'affichage et la somme des matrices :

### **Algorithme Matrices**

variables **tableau** M1[3][4],M2 [3][4],M3 [3][4] :: réel

#### **Début**

SaisieMatrice(3, 4, M1)

SaisieMatrice(3, 4, M2)

AfficheMatrice(3,4, M1)

AfficheMatrice(3,4, M2)

SommeMatrice(3, 4, M1,M2,M3)

AfficheMatrice(3,4, M3)

#### **Fin**

# Tableaux : trouver l'erreur

## Algorithme 1

**Essai de tableau 1**

**Variables**

i : entier

Tab(10) : tableau d'entiers

**Début**

**tant que** (i <= 10)

    lire tab(i)

    i  $\leftarrow$  i+1

**Fin de tant que**

**Fin**

## Algorithme 2

**Essai de tableau 2**

**Variables**

i, x : entiers

Tab(10) : tableau d'entiers

**Début**

**pour** i variant de 1 à 10

**Si** Tab(i+1) < Tab(i) **alors**

        permute Tab(i), Tab(i+1)

**Fin de Si**

**Fin de pour**

**Fin**

## Tableaux : Exemple d'exercice

---

Écrire l'algorithme du traitement qui permet de saisir 10 nombres entiers dans un tableau à une dimension, puis qui recherche et affiche la valeur minimale entrée dans un tableau. L'affichage mentionnera également l'indice auquel se trouve ce minimum.

# Algorithme recherche Mini

---

**Recherche Mini**

**Variables**

**i, Indice\_Mini, Mini : entiers**

**Tab(10) : tableau d'entiers**

**Début**

**pour i variant de 1 à 10**

**Lire Tab(i)**

**Fin de pour**

**Mini ← Tab(1)**

**Indice\_Mini ← 1**

**pour i variant de 2 à 10**

**Si Tab(i) < Mini alors**

**Mini ← Tab(i)**

**Indice\_Mini ← i**

**Fin de Si**

**Fin de pour**

**ecrire (Mini, Indice\_Mini)**

**Fin**

# Tableaux : 2 problèmes classiques

---

- **Recherche d'un élément dans un tableau**
  - Recherche séquentielle
  - Recherche dichotomique
- **Tri d'un tableau**
  - Tri par sélection
  - Tri rapide

# Recherche séquentielle

- Recherche de la valeur x dans un tableau T de N éléments :

**Variables** i: entier, Trouvé : booléen

...

i←0 , Trouvé ← Faux

**TantQue** (i < N) ET (Trouvé=Faux)

**Si** (T[i]=x) **alors**

        Trouvé ← Vrai

**Sinon**

        i←i+1

**FinSi**

**FinTantQue**

**Si** Trouvé **alors** // c'est équivalent à écrire **Si** Trouvé=Vrai **alors**

        écrire ("x appartient au tableau")

**Sinon**écrire ("x n'appartient pas au tableau")

**FinSi**

## **Recherche séquentielle (version 2)**

---

- Une fonction Recherche qui retourne un booléen pour indiquer si une valeur x appartient à un tableau T de dimension N.  
x , N et T sont des paramètres de la fonction

**Fonction** Recherche(x : réel, N: entier, tableau T : réel ) : booléen

**Variable** i: entier

**Pour** i allant de 0 à N-1

**Si** (T[i]=x) **alors**

    retourne (Vrai)

**FinSi**

**FinPour**

    retourne (Faux)

**FinFonction**

# Notion de complexité d'un algorithme

---

- Pour évaluer l'**efficacité** d'un algorithme, on calcule sa **complexité**
- Mesurer la **complexité** revient à quantifier le **temps** d'exécution et l'espace **mémoire** nécessaire
- Le temps d'exécution est proportionnel au **nombre des opérations** effectuées. Pour mesurer la complexité en temps, on met en évidence certaines opérations fondamentales, puis on les compte
- Le nombre d'opérations dépend généralement du **nombre de données** à traiter. Ainsi, la complexité est une fonction de la taille des données. On s'intéresse souvent à son **ordre de grandeur** asymptotique
- En général, on s'intéresse à la **complexité dans le pire des cas** et à la **complexité moyenne**

## Recherche séquentielle : complexité

- Pour évaluer l'efficacité de l'algorithme de recherche séquentielle, on va calculer sa complexité dans le pire des cas. Pour cela on va compter le nombre de tests effectués
- Le pire des cas pour cet algorithme correspond au cas où  $x$  n'est pas dans le tableau  $T$
- Si  $x$  n'est pas dans le tableau, on effectue  $3N$  tests : on répète  $N$  fois les tests  $(i < N)$ ,  $(\text{Trouvé}=\text{Faux})$  et  $(T[i]=x)$
- La **complexité** dans le pire des cas est **d'ordre  $N$** , (on note  **$O(N)$** )
- Pour un ordinateur qui effectue  $10^6$  tests par seconde on a :

$N$	$10^3$	$10^6$	$10^9$
temps	1ms	1s	16mn40s

# Recherche dichotomique

---

- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique
- **Principe :** diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur  $x$  à chaque étape de la recherche. Pour cela on compare  $x$  avec  $T[\text{milieu}]$  :
  - Si  $x < T[\text{milieu}]$ , il suffit de chercher  $x$  dans la 1ère moitié du tableau entre ( $T[0]$  et  $T[\text{milieu}-1]$ )
  - Si  $x > T[\text{milieu}]$ , il suffit de chercher  $x$  dans la 2ème moitié du tableau entre ( $T[\text{milieu}+1]$  et  $T[N-1]$ )

# Recherche dichotomique : algorithme

inf←0 , sup←N-1, Trouvé ← Faux

**TantQue** (inf <=sup) ET (Trouvé=Faux)

    milieu←(inf+sup)div2

**Si** (x=T[milieu]) **alors**

        Trouvé ← Vrai

**SinonSi** (x>T[milieu]) **alors**

        inf←milieu+1

**Sinon** sup←milieu-1

**FinSi**

**FinSi**

**FinTantQue**

**Si** Trouvé **alors**   **écrire** ("x appartient au tableau")

**Sinon**                   **écrire** ("x n'appartient pas au tableau")

**FinSi**

# Exemple d'exécution

- Considérons le tableau T : 4 6 10 15 17 18 24 27 30
- Si la valeur cherché est 20 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	5	5	6
sup	8	8	5	5
milieu	4	6	5	

- Si la valeur cherché est 10 alors les indices inf, sup et milieu vont évoluer comme suit :

inf	0	0	2
sup	8	3	3
milieu	4	1	2

## **Recherche dichotomique : complexité**

---

- La complexité dans le pire des cas est d'ordre  $\log_2 N$
- L'écart de performances entre la recherche séquentielle et la recherche dichotomique est considérable pour les grandes valeurs de  $N$
- Exemple: au lieu de  $N=1\text{million} \approx 2^{20}$  opérations à effectuer avec une recherche séquentielle il suffit de 20 opérations avec une recherche dichotomique

# Algorithmique

*Les méthodes de Tris*

# Tris d'un tableau

Pour pouvoir comparer les méthodes de tris, on les appliquent à des tableaux d'entiers.

Tableau non trié :

20	6	1	3	1	7
----	---	---	---	---	---

Indices : 0 1 2 3 4 5

Tableau trié :

1	1	3	6	7	20
---	---	---	---	---	----

Indices : 0 1 2 3 4 5

**Constante N <- ...**

**Programme de tri**

**Algorithme**

**Début**

Type tab = tableau[ N ]: entier

Variable t: tab

tri( t )

**Fin**

**Procédure tri( t: tab en entrée sortie )**

**Algorithme**

**Début**

...

**Fin**

La méthode de tri

# **Les algorithmes de Tri**

---

- Il existe plusieurs algorithmes connus pour trier les éléments d'un tableau :
  - Tris élémentaires
    - Le tri par sélection
    - Le tri par insertion
    - Le tri à bulles
  - Tris avancées
    - Tri rapide
    - .....

# Tri par sélection

Méthode :

on cherche l'élément de plus petite valeur pour l'échanger avec l'élément en première position ;

puis on cherche l'élément ayant la deuxième plus petite valeur pour l'échanger avec l'élément en deuxième position ;

et ainsi de suite.

20	6	1	3	1	7
1	6	20	3	1	7
1	1	20	3	6	7
1	1	3	20	6	7
1	1	3	6	20	7
1	1	3	6	7	20

Il faut :

- 1 boucle pour parcourir le tableau et sélectionner tous les éléments ;
- 1 boucle pour rechercher le minimum parmi les éléments non triés.

# Tri par sélection : méthodologie

---

- Nous avions une certaine situation sur l'intervalle  $[ 0 .. i-1 ]$  :
  - les éléments jusqu'à  $i-1$  sont triés,
  - ceux qui suivent sont plus grands, mais pas triés.
- Nous retrouvons la même situation sur l'intervalle  $[ 0 .. i ]$  :
  - les éléments jusqu'à  $i$  sont triés,
  - ceux qui suivent sont plus grands, mais pas triés.
- Cette propriété est donc invariante avec  $i$ , on l'appelle

# Tri par sélection : algorithme

- Supposons que le tableau est noté T et sa taille N

**Pour** i allant de 0 à N-2

    indice  $\leftarrow$  i

**Pour** j allant de i + 1 à N-1

**Si** T[j] < T[indice] **alors**

        indice  $\leftarrow$  j

**Finsi**

**FinPour**

temp  $\leftarrow$  T[indice]

    T[indice]  $\leftarrow$  T[i]

    T[i]  $\leftarrow$  temp

**FinPour**



Réservation de la case



Recherche de l'élément  
concerné



Permutation des deux valeurs

## Tri par sélection : exemple

---

- Simuler l'algorithme du tri par sélection sur le tableau suivant en montrant les étapes d'exécution.

9     4     1     7     3

# Tri par sélection : exemple

- **Principe** : à l'étape  $i$ , on sélectionne le plus petit élément parmi les  $(n - i + 1)$  éléments du tableau les plus à droite. On l'échange ensuite avec l'élément  $i$  du tableau
- **Exemple** :

9     4     1     7     3

- **Étape 1:** on cherche le plus petit parmi les 5 éléments du tableau. On l'identifie en troisième position, et on l'échange alors avec l'élément 1 :

1     4     9     7     3

- **Étape 2:** on cherche le plus petit élément, mais cette fois à partir du deuxième élément. On le trouve en dernière position, on l'échange avec le deuxième:

1     3     9     7     4

- **Étape 3:**

1     3     4     7     9

# Tri par sélection : complexité

Procédure triSélection( t: tab en entrée sortie )

Algorithme

Début

Pour i de 0 à N-2 répéter

/\* boucle N-1 fois \*/

Pour j de i+1 à N-1 répéter

/\* boucle N-i fois \*/

Si t[ j ] < t[ min ] alors

/\*  $(N-1)+(N-2)+\dots+2+1=N(N-1)/2$  comparaisons \*/

min <- j

/\* seule partie qui dépend des données \*/

Fin si

Fin pour

...

/\* N-1 échanges \*/

Fin pour

Fin



Tri par sélection :

- nombre de comparaison de l'ordre de  $N^2/2$  ;
- nombre d'échanges de l'ordre de N.

# Tri par insertion

Méthode :

on considère les éléments les uns après les autres en insérant chacun à sa place parmi les éléments déjà triés.

20	6	1	3	1	7
6	20	1	3	1	7
1	6	20	3	1	7
1	3	6	20	1	7
1	1	3	6	20	7
1	1	3	6	7	20

Il faut :

- 1 boucle pour parcourir le tableau et sélectionner l'élément à insérer ;
- 1 boucle pour décaler les éléments plus grands que l'élément à insérer ;
- insérer l'élément.

# Tri par insertion : algorithme

**Procédure triInsertion( t: tab en entrée sortie )**

## Algorithme

## Début

Variables i, j, mem: entier

Pour i de 1 à N-1 répéter

/\* sélection de l'élément à insérer \*/

```
mem <- t[i]
```

$j \leftarrow i$

Tant que  $j > 0$  et  $t[j-1] > \text{mem}$  répéter /\* décalage des éléments plus grands \*/

$t[j] \leftarrow t[j-1]$

$j \leftarrow j - 1$

**Fin tant que**

```
t[j] <- mem
```

/\* insertion \*/

**Fin pour**

Fin

# Tri par insertion : exemple

---

- Simuler l'algorithme du tri par insertion sur le tableau suivant en montrant les étapes d'exécution.

2        56        4        7        0

# Tri par insertion : Exemple

---

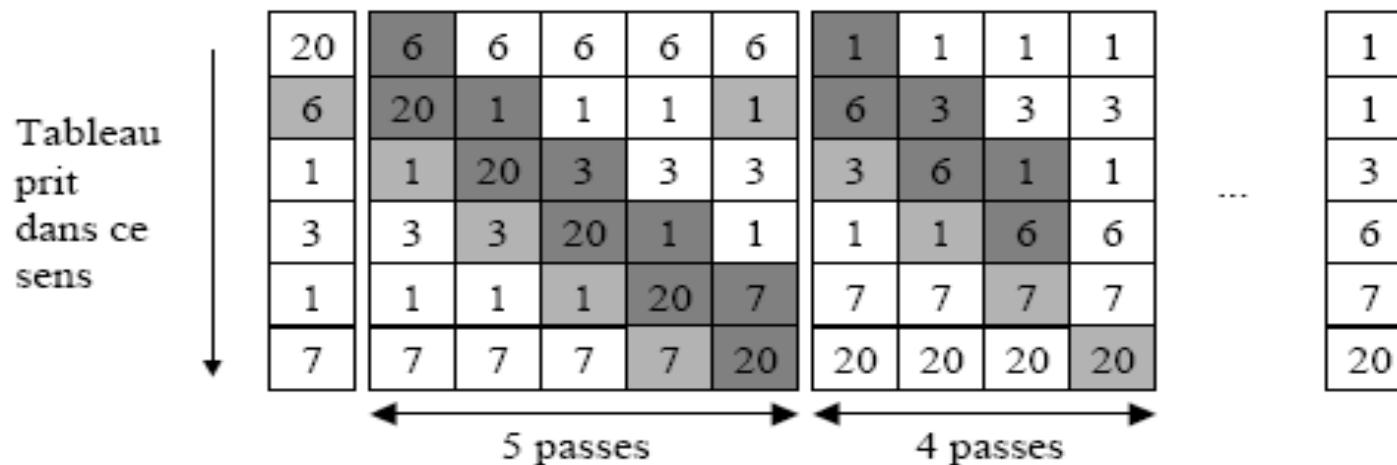
- Prendre l'élément i
  - Insérer i dans l'ordre entre 0 et i
  - Continuer à partir de i+1

- 2, 56, 4, 7, 0
- 2, 56, 4, 7, 0
- 2, 56, 4, 7, 0
- 2, 4, 56, 7, 0
- 2, 4, 7, 56, 0
- 0, 2, 4, 7, 56

# Tri à bulles

Méthode :

on parcours autant de fois le tableau en permutant 2 éléments adjacents mal classés qu'il le faut pour que le tableau soit trié.



Il faut :

- 1 boucle pour parcourir tout le tableau et sélectionner les éléments un à un ;
- 1 boucle pour permutez les éléments adjacents.

# Tri à bulles : algorithme

Procédure triBulles( t: tab en entrée sortie )

Algorithme

Début

Variables i, j, temp: entier

Pour i de N-1 à 0 répéter /\* sélection de l'élément à insérer \*/

    Pour j de 1 à i répéter /\* décalage des éléments plus grands \*/

        Si t[ j-1 ] > t[ j ] alors

            tmp <- t[ j-1 ] /\* échange \*/

            t[ j-1 ] <- t[ j ]

            t[ j ] <- tmp

    Fin si

Fin pour

Fin pour

Fin

## Tri à bulles : exemple

---

- Simuler l'algorithme du tri à bulles sur le tableau suivant en montrant les étapes d'exécution.

4            1            5            3            2

# Tri à bulles : exemple

Un balayage

Sens de  
lecture de  
tableau

2
3
5
1
4

2
3
5
1
4

2
3
1
5
4

2
1
3
5
4

1
2
3
5
4

Suite des balayages

2
3
5
1
4

1
2
3
5
4

1
2
3
4
5

1
2
3
4
5

1
2
3
4
5

peuvent être éliminés

# Tri rapide

---

- Le tri rapide est un tri récursif basé sur l'approche "diviser pour régner" (consiste à décomposer un problème d'une taille donnée à des sous problèmes similaires mais de taille inférieure faciles à résoudre)
- **Description du tri rapide :**
  - **1)** on considère un élément du tableau qu'on appelle pivot
  - **2)** on partitionne le tableau en 2 sous tableaux : les éléments inférieurs ou égaux à pivot et les éléments supérieurs à pivot. on peut placer ainsi la valeur du pivot à sa place définitive entre les deux sous tableaux
  - **3)** on répète récursivement ce partitionnement sur chacun des sous tableaux créés jusqu'à ce qu'ils soient réduits à un à un seul élément

# Tri Rapide

Méthode diviser pour résoudre :

- ① choisir un seuil (pivot) ;
- ② mettre les éléments plus petit que le seuil à gauche du seuil et les éléments plus grands à droite ;
- ③ recommencer séparément sur les parties droite et gauche.

①	20	6	1	3	1	7
②	1	6	1	3	7	20
③	1		6	1	3	
①	1		6	1	3	
②	1		3	6		

Il faut :

- partitionner selon un pivot ;
- recommencer sur les partitions (récursion).

# Procédure Tri rapide

**Procédure** TriRapide(tableau **T** : réel par adresse, **p,r**: entier par valeur)

variable **q**: entier

**Si** **p < r alors**

    Partition(**T,p,r,q**)

    TriRapide(**T,p,q-1**)

    TriRapide(**T,q+1,r**)

**FinSi**

**Fin Procédure**

A chaque étape de récursivité on partitionne un tableau  $T[p..r]$  en deux sous tableaux  $T[p..q-1]$  et  $T[q+1..r]$  tel que chaque élément de  $T[p..q-1]$  soit inférieur ou égal à chaque élément de  $A[q+1..r]$ . L'indice **q** est calculé pendant la procédure de partitionnement

# Procédure de partition

**Procédure** Partition(tableau **T** : réel par adresse, **p,r**: entier par valeur,  
**q**: entier par adresse )

Variables i, j: entier

pivot: réel

$\text{pivot} \leftarrow T[p], i \leftarrow p+1, j \leftarrow m$

## TantQue ( $i \leq j$ )

**TantQue** ( $i \leq r$  et  $T[i] \leq pivot$ )     $i \leftarrow i+1$  **FinTantQue**

**TantQue** ( $j \geq p$  et  $T[j] > \text{pivot}$ )    $j \leftarrow j-1$    **FinTantQue**

**Si  $i < j$  alors**

Echanger( $T[i]$ ,  $T[j]$ ),  $i \leftarrow i+1$ ,  $j \leftarrow j-1$

FinSi

FinTantQue

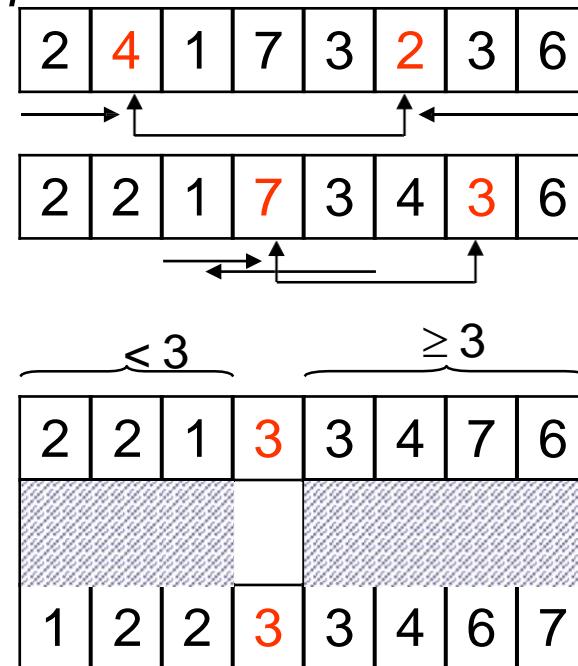
Echanger( $T[j]$ ,  $T[p]$ )

$q \uparrow j$

## **Fin Procédure**

# Tri Rapide : Exemple

Partage avec  $pivot = 3$



Suite du tri

## Tri rapide : complexité et remarques

---

- La complexité du tri rapide dans le pire des cas est en  $O(N^2)$
- La complexité du tri rapide en moyenne est en  $O(N \log N)$
- Le choix du pivot influence largement les performances du tri rapide
- Le pire des cas correspond au cas où le pivot est à chaque choix le plus petit élément du tableau (tableau déjà trié)