Université de Batna 2

Faculté de technologie

Département D'hydraulique

Hydro-informatique

A.AMEZIANE

3ème Licence Hydraulique

Semestre 6

2019/2020

Objectifs de l'enseignement et plan du cours

- **Objectif:**
  - Apprendre les concepts de base de l'algorithmique et de la programmation
  - Etre capable de maîtriser les différentes techniques de l'analyse numérique dans le domaine de la modélisation en hydraulique, hydrologie, hydraulique fluviale.
  - Mettre en œuvre ces concepts pour écrire les programmes correspondants.

- **Plan:**
  - Généralités (systèmes d'exploitation, langages de programmation, …)
  - Chapitre 1. Organigrammes et algorithmes (affectation, instructions conditionnelles, instructions itératives, fonctions, procédures, …)
  - Chapitre 2. Programme des applications numériques (Etude du langage Fortran avec quelque méthodes numériques)
  - Chapitre 3. Programmation en hydraulique en FORTRAN et MATLAB (conception et programmation en fortran d'un cas en hydraulique)
  - Chapitre 4. Projets de cours/ Devoirs: Résolution numérique par MATLAB et FORTRAN d'un problème en hydraulique, (et/ou) en hydrologie.

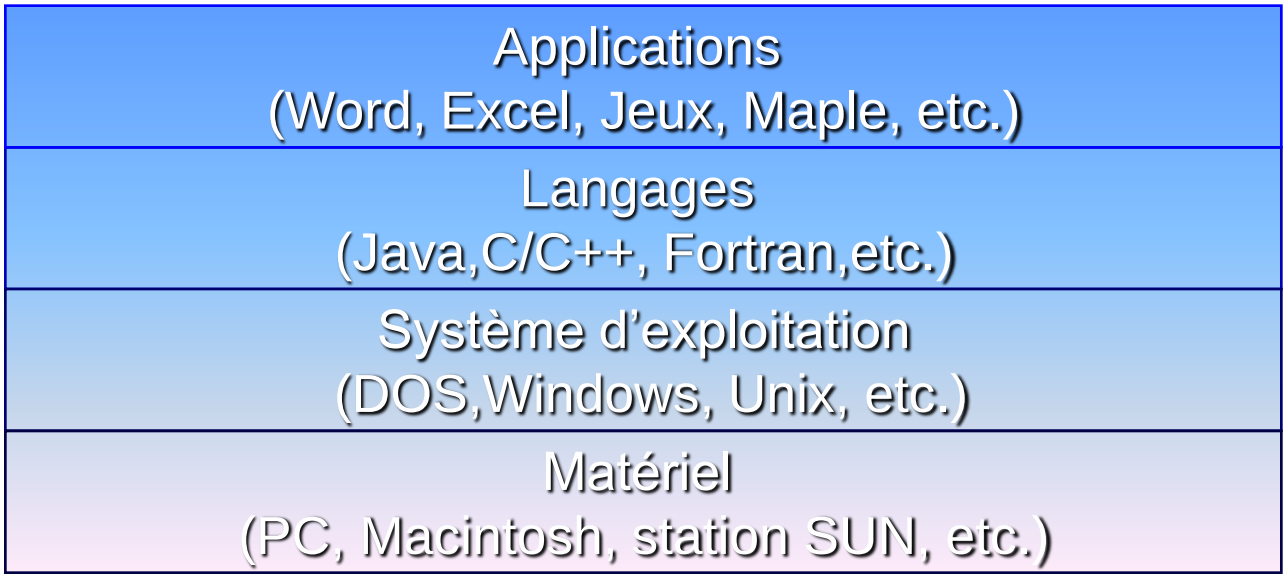
Généralités

Notions d'informatique

Qu'est-ce que l'informatique?

- L'informatique : Est la science du traitement **automatique** de l'**information** au moyen des machines intelligentes (Ordinateurs).

- Eléments d'un système informatique



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

- Ensemble de modules (programmes) qui gèrent le matériel (la partie Hardware) et contrôlent les applications (la partie Software)
  - Gestion des périphériques (affichage à l'écran, lecture du clavier, pilotage d'une imprimante, …)
  - Gestion des processus et de leurs ressources (comptes, partage des ressources, gestion des fichiers et répertoires, …)
  - Interface homme machine « avec l'utilisateur » (textuelle ou graphique): Interprétation des commandes
  - Contrôle des programmes (découpage en taches, partage du temps processeur, …)

Langages de programmation

- 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) déstinés à effectuer une tache donnée
  - Exemple: un programme de gestion de comptes bancaires
- Contrainte: être compréhensible par la machine

Langage machine

- Langage **binaire**: l'information est exprimée et manipulée sous forme d'une suite de bits
- Un **bit** (*binary digit*) = 0 ou 1 (2 états électriques)
- Une combinaison de 8 bits= 1 **Octet** ➔ $2^8 = 256$ possibilités qui permettent de coder tous les caractères alphabétiques, numériques, et symboles tels que ?,*,&, …
  - 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

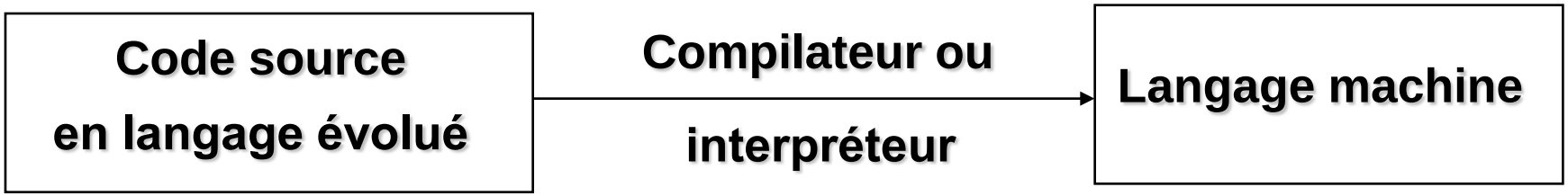
ADD A, 4
LOAD B → traducteur → langage machine
MOV A, OUT
...

  - +: 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



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 du programme à chaque exécution

  - + exécution instantanée appréciable pour les débutants
  - - exécution lente par rapport à la compilation

Langages de programmation:

- Deux types de langages:
  - Langages procéduraux
  - Langages orientés objets
- Exemples de langages:
  - **Fortran, Matlab;Cobol, Pascal, C, …**
  - **C++, Java, …**
- Choix d'un langage?

Etapes de réalisation d'un programme

Enoncé du problème

↓ *Spécification*

Cahier des charges

↓ *Analyse*

Algorithme

↓ *Traduction en langage*

Programme source

↓ *Compilation*

Programme exécutable

↓ *Tests et modifications*

Version finale et résultats

La réalisation de programmes passe par l'écriture d'algorithmes

$\Rightarrow$ D'où l'intérêt de l'**Algorithmique**

Chapitre 1 : Organigrammes et algorithmes

Notions fondamentales en organigramme et algorithmique

Algorithmique

- Le terme **algorithme** vient du nom du mathématicien arabe **Al-Khawarizmi** (820 après J.C.)
- 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)
  - Qualités: **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), …
- **L'algorithmique** désigne aussi la discipline qui étudie les algorithmes et leurs applications en Informatique
- Une bonne connaissance de l'algorithmique permet d'écrire des algorithmes exacts et efficaces

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

Organigrammes

Notions et structures de base

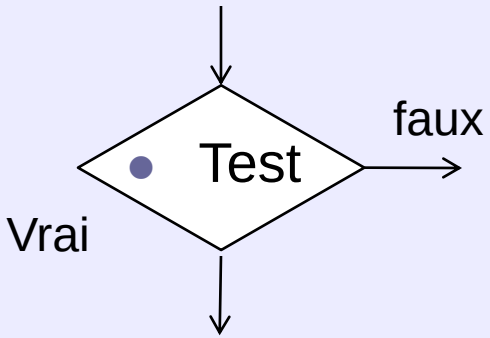
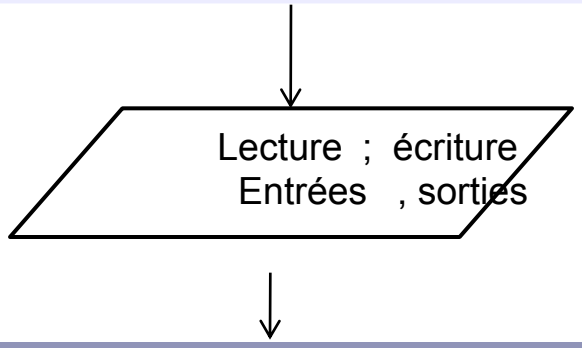
Organigramme et la norme ISO

- Un **organigramme de programmation** (parfois appelé algorigramme , **logigramme** ou plus rarement **ordinogramme**) est une représentation graphique normalisée de l'enchaînement des opérations et des décisions effectuées par un programme d'ordinateur.

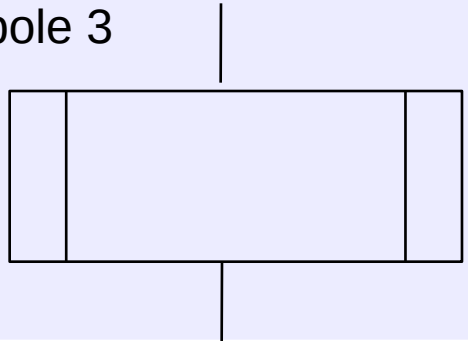
- La norme ISO 5807 décrit en détail les différents symboles à utiliser pour représenter un programme informatique de manière normalisée

Les symboles de l'organigramme

- L'organigramme de programmation utilise des symboles normalisés représentés ci-dessous :

Symboles Normalisés	commentaires
	Les tests ou branchements conditionnels : • l'entrée du test est l'angle du haut ; • la sortie avec le rond est le résultat du test lorsqu'il est faux ; • la sortie sans rond est le résultat du test lorsqu'il est vrai.
	Mise à disposition d'une information à traiter ou enregistrement d'une information traitée.

Les symboles de l'organigramme

Symboles Normalisés	commentaires
Symbole 3 	Appel de sous-programme.

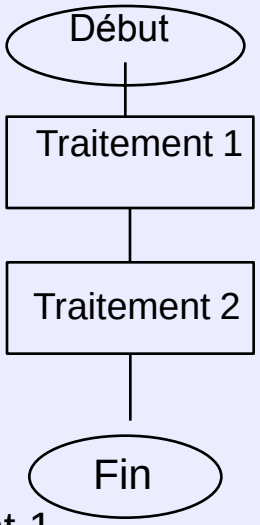
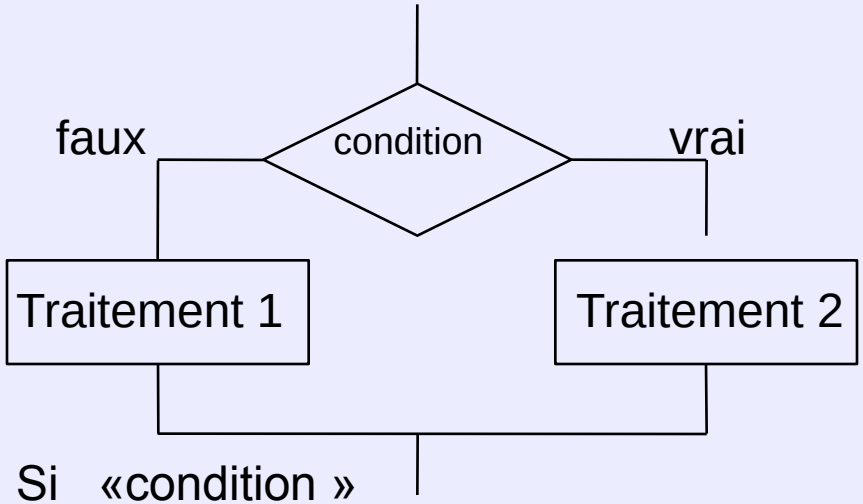
Le sens par défaut des liens du flux d'exécution est :

- du haut vers le bas pour les liens verticaux ;
- de la gauche vers la droite pour les liens horizontaux.

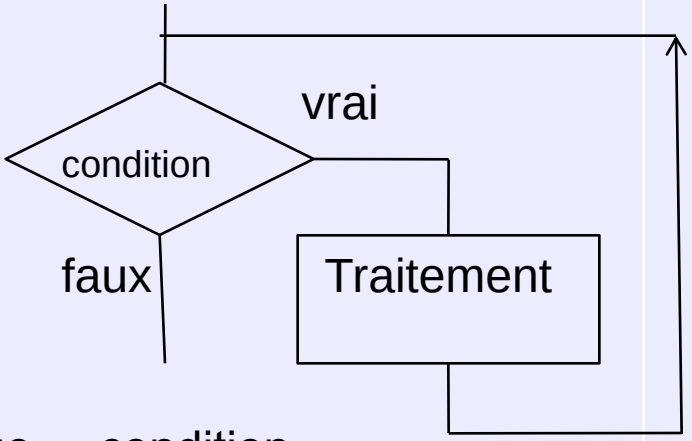
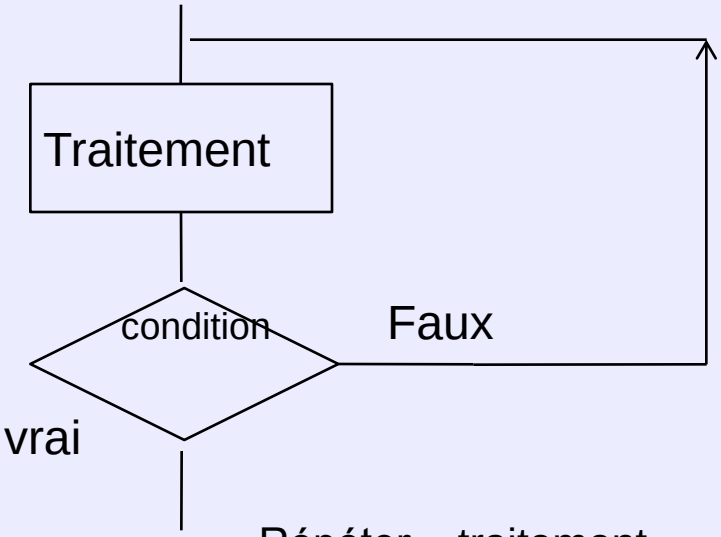
Lorsque le sens par défaut n'est pas respecté, il est nécessaire de le préciser par une flèche à l'extrémité du lien.

Les différentes structures de l'organigramme de programmation

- L'organigramme de programmation utilise les structures représentés ci-dessous :

Séquence linéaire	Séquence alternative« si…alors…sinon »
 debut • traitement 1 • traitement 2 fin	 Si «condition » alors : «Traitement 1 » sinon : « traitement 2 » finsi

Les différentes structures de l'organigramme de programmation

Séquence répétitive« tant que…faire… »	Séquence répétitive« répéter…jusqu'à… »
	
Tant que « condition » * faire : « traitement » Fin tant que	Répéter « traitement » jusqu'à « condition »

Algorithmique

Notions et instructions de base

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:

- Un nom doit commencer par une lettre alphabétique
  exemple valide: S1 **exemple invalide: 1S**
- doit être constitué uniquement de lettres, de chiffres et du soulignement _ (Eviter les caractères de ponctuation et les espaces)
  valides: HYD200, HYD_2020 **invalides: HYD 2020,HYD-2020,HYD;2020**
- doit être différent des mots réservés du langage (par exemple en Java: **int, float, else, switch, case, default, for, main, return**, …)
- La longueur du nom doit être inférieure à la taille maximale spécifiée par le langage utilisé

Choix des identificateurs (2)

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

exemples: force ,moment_inertie, vitesse

Remarque: en pseudo-code algorithmique, 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 1octet): 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, Ville", "code Commune: 3650", …

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 Max ,N,k : entier

x1, X2 : réel

trouver: booléen

str1, str2 : 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 ←

  Var← 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:** **max ←1** **N ←i** **k ←max+N**
  x 1←10.3 **trouver←FAUX** **str11 ←"SMI"**
  str2 ←str1 **x1 ←4** **x1 ←j**

  (voir la déclaration des variables dans le transparent précédent)

- **non valides:** **Max ←10.3** **trouver←« algeria«** **k ←x1**

Quelques remarques

- Beaucoup de langages de programmation (C/C++, Java, …) utilisent le signe égal = pour l'affectation ←. 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

Expressions et opérateurs

- Une **expression** peut être une valeur, une variable ou une opération constituée de variables reliées par des **opérateurs**
- 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**: **=, $\neq$, <, >, <=, >=**
  - **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) :
  - ^ : (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

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

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

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**
VRAI	VRAI	**VRAI**
VRAI	FAUX	**FAUX**
FAUX	VRAI	**FAUX**
FAUX	FAUX	**FAUX**

C1	C2	**C1 OU C2**
VRAI	VRAI	**VRAI**
VRAI	FAUX	**VRAI**
FAUX	VRAI	**VRAI**
FAUX	FAUX	**FAUX**

C1	C2	**C1 XOR C2**
VRAI	VRAI	**FAUX**
VRAI	FAUX	**VRAI**
FAUX	VRAI	**VRAI**
FAUX	FAUX	**FAUX**

C1	**NON C1**
VRAI	**FAUX**
FAUX	**VRAI**

Tests imbriqués « hiérarchiques »

- 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
```

Instructions itératives: les boucles

- Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois
- On distingue trois sortes de boucles en langages de programmation :
  - Les **boucles tant que** : on y répète des instructions tant qu'une certaine condition est réalisée
  - Les **boucles jusqu'à** : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
  - Les **boucles pour** ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

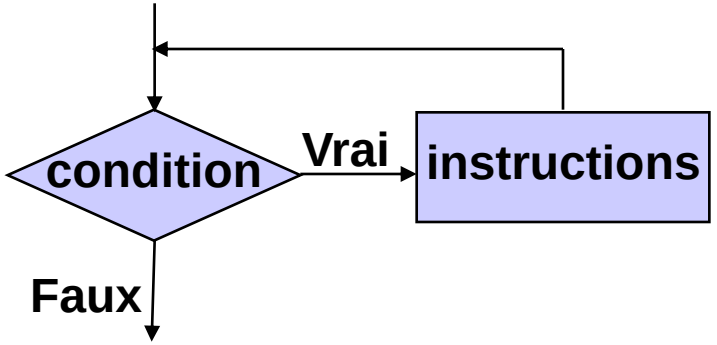
(Dans ce cours, on va s'intéresser essentiellement aux boucles *Tant que* et boucles *Pour* qui sont plus utilisées et qui sont définies en Maple)

Les boucles Tant que

TantQue (condition)

instructions

FinTantQue



- la condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, …
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après FinTantQue

Les boucles Tant que : remarques

- Le nombre d'itérations dans une boucle TantQue n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition

- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment

$\Rightarrow$ **Attention aux boucles infinies**

- Exemple de boucle infinie :

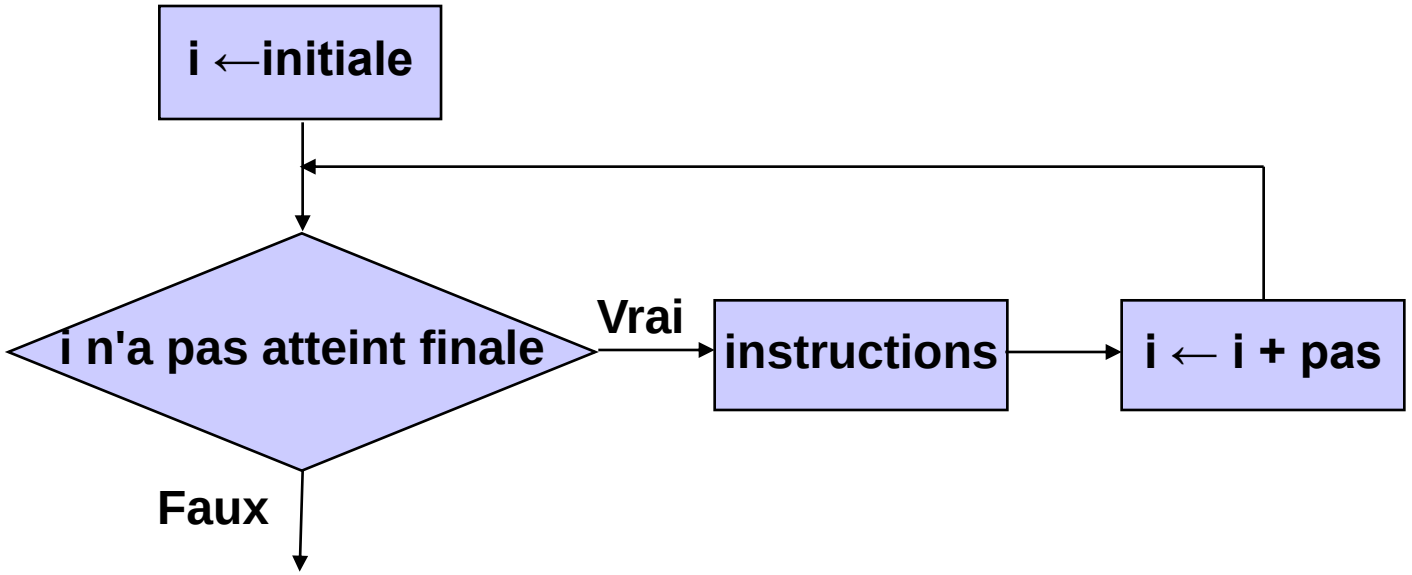
```
i ← 2
TantQue   (i > 0)
i ←  i+1        (attention aux erreurs de frappe : + au lieu de -)
FinTantQue
```

Les boucles Pour

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

instructions

FinPour



Les boucles Pour

- Remarque : le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle

- **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 <= à finale dans le cas d'un pas positif (ou si compteur est >= 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 : 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** 10 **à** 100
i ← 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 ← initiale
TantQue compteur <= finale
instructions
compteur ← compteur+pas
FinTantQue

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
```

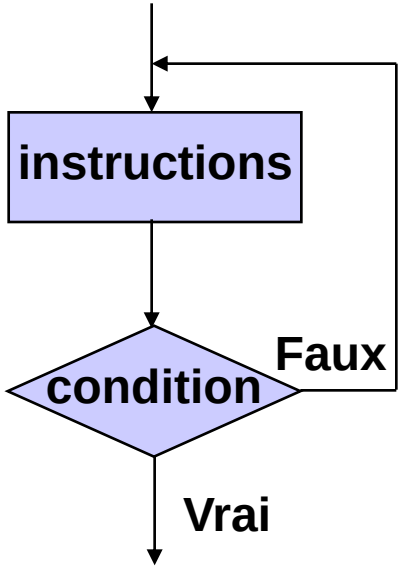
Exécution

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 ← 0
        i ← 0
        Répéter
                i ← i+1
                som ← 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*

Programmation Modulaire

Fonctions et procédures

Fonctions et procédures

- Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :
  - permettent de **"factoriser" les programmes**, càd de mettre en commun les parties qui se répètent
  - permettent une **structuration** et une **meilleure lisibilité** des programmes
  - **facilitent la maintenance** du code (il suffit de modifier une seule fois)
  - ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes

Fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat à partir des valeurs des paramètres**
- Une fonction s'écrit en dehors du programme principal sous la forme :

Fonction nom_fonction (paramètres et leurs types) : type_fonction

    Instructions constituant le corps de la fonction

    retourne …

FinFonction

- Pour le choix d'un nom de fonction il faut respecter les mêmes règles que celles pour les noms de variables
- type_fonction est le type du résultat retourné
- L'instruction **retourne** sert à retourner la valeur du résultat

Fonctions : exemples

- La fonction SommeCarre suivante calcule la somme des carrées de deux réels x et y :

```
Fonction SommeCarre (x : réel, y: réel ) : réel
        variable z : réel
        z ←x^2+y^2
        retourne (z)
FinFonction
```

- La fonction Pair suivante détermine si un nombre est pair :

```
Fonction  Pair (n : entier ) : booléen
        retourne (n%2=0)
FinFonction
```

Utilisation des fonctions

- L'utilisation d'une fonction se fera par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, devra être affecté ou être utilisé dans une expression, une écriture, ...

- **Exepmle :** **Algorithme exepmleAppelFonction**

```
variables     z : réel,  b : booléen
Début
    b ←Pair(3)
    z ←5*SommeCarre(7,2)+1
    écrire("SommeCarre(3,5)= ", SommeCarre(3,5))
Fin
```

- Lors de l'appel Pair(3) le **paramètre formel** n est remplacé par le **paramètre effectif** 3

Procèdures

- Dans certains cas, on peut avoir besoin de répéter une tache dans plusieurs endroits du programme, mais que dans cette tache on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois
- Dans ces cas on ne peut pas utiliser une fonction, on utilise une **procédure**
- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**
- Une procédure s'écrit en dehors du programme principal sous la forme :

Procédure nom_procédure (paramètres et leurs types)

Instructions constituant le corps de la procédure

FinProcédure

- Remarque : une procédure peut ne pas avoir de paramètres

Appel d'une procédure

- L'appel d'une procédure, se fait dans le programme principale ou dans une autre procédure par une instruction indiquant le nom de la procédure :

```
Procédure exemple_proc (…)
        …
FinProcédure

Algorithme exepmleAppelProcédure
Début
        exemple_proc (…)
        …
Fin
```

- Remarque : contrairement à l'appel d'une fonction, on ne peut pas affecter la procédure appelée ou l'utiliser dans une expression. L'appel d'une procédure est une instruction autonome

Paramètres d'une procédure

- Les paramètres servent à échanger des données entre le programme principale (ou la procédure appelante) et la procédure appelée
- Les paramètres placés dans la déclaration d'une procédure sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)
- Les paramètres placés dans l'appel d'une procédure sont appelés **paramètres effectifs**. ils contiennent les valeurs pour effectuer le traitement
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre

Passage des paramètres

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

- **Le passage 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 passage 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

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 ou 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 principale. 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 ← 0

Si (N1 >10) alors nbre ←nbre+1 FinSi

….

Si (N30>10) alors nbre ←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 : 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

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 : 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), (Trouvé=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[milieu] :

  - Si x < T[milieu], il suffit de chercher x dans la 1ère moitié du tableau entre (T[0] et T[milieu-1])

  - Si x > T[milieu], il suffit de chercher x dans la 2ème moitié du tableau entre (T[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=1milion ≈2^{20} opérations à effectuer avec une recherche séquentielle il suffit de 20 opérations avec une recherche dichotomique

Tri d'un tableau

- Le tri consiste à ordonner les éléments du tableau dans l'ordre croissant ou décroissant

- Il existe plusieurs algorithmes connus pour trier les éléments d'un tableau :

  - Le tri par sélection
  - Le tri par insertion
  - Le tri rapide
  - …

- Nous verrons dans la suite l'algorithme de tri par sélection et l'algorithme de tri rapide. Le tri sera effectué dans l'ordre croissant

Tri par sélection

- **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 : algorithme

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

```
Pour i  allant de 0 à N-2
        indice_ppe ← i
        Pour j allant de i + 1 à N-1
            Si T[j] <T[indice_ppe] alors
                indice_ppe  ← j
            Finsi
        FinPour

        temp ← T[indice_ppe]
        T[indice_ppe] ← T[i]
        T[i] ← temp
FinPour
```

Tri par sélection : complexité

- Quel que soit l'ordre du tableau initial, le nombre de tests et d'échanges reste le même

- On effectue N-1 tests pour trouver le premier élément du tableau trié, N-2 tests pour le deuxième, et ainsi de suite. Soit : (N-1)+(N-2)+…+1 = N(N-1)/2
  On effectue en plus (N-1) échanges.

- La **complexité** du tri par sélection est **d'ordre N²** à la fois dans le meilleur des cas, en moyenne et dans le pire des cas

- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1s	11,5 jours	32000 ans

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 crées jusqu'à ce qu'ils soient réduits à un à un seul élément

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
     pivot← T[p], i←p+1, j ← r
     TantQue (i<=j)
          TantQue (i<=r et T[i] <=pivot)     i ← i+1  FinTantQue
          TantQue (j>=p et T[j] >pivot )  j ← j-1    FinTantQue
          Si  i <j  alors
                    Echanger(T[i], T[j]), i ← i+1, j ← j-1
          FinSi
     FinTantQue
      Echanger(T[j], T[p])
     q ← j
Fin Procédure
```

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)
- différentes versions du tri rapide sont proposés dans la littérature pour rendre le pire des cas le plus improbable possible, ce qui rend cette méthode la plus rapide en moyenne parmi toutes celles utilisées

Chapitre 2 : Etude d'un Langage de programmation Numérique

FORTRAN