



*Institut Supérieur d'Administration des Entreprises (**ISAE**) THIES*



*Module de Première Année
Informatique de Gestion, Semestre 2*

Conception et Analyse d'Algorithmes

Objectifs

- Acquérir une démarche pour trouver une solution informatique à un problème donné
- 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
- Faire comprendre aux étudiants:
 - comment fonctionne un programme informatique
 - que l'algorithme est l'ultime étape avant l'écriture des codes programmes

Sommaire

- **Partie 1** : Notions et Instructions de base
- **Partie 2** : Les Structures alternatives (Tests)
- **Partie 3** : Les Structures répétitives (Boucles)
- **Partie 4** : Les Tableaux et les Algorithmes de Tri
- **Partie 5** : Les Fonctions Prédéfinies/ Procédures et Fonctions
- **Partie 6** : Les Variables Structurées et Les Fichiers

Partie 1

Notions et instructions de base

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

Les qualités requises

La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs :

- il faut avoir une certaine intuition.
- il faut être méthodique et rigoureux.

Evitez de sauter les étapes : la vérification méthodique, pas à pas, de chacun de vos algorithmes représente plus de la moitié du travail à accomplir.

Les catégories d'instructions

Les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'instructions:

- L'affectation de variable
- La lecture / écriture
- Les tests
- Les boucles

Un algorithme informatique se ramène donc toujours à la combinaison de ces quatre instructions de base.

Algorithmique et programmation

- L'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.
- Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique.
- Lorsqu'on programme dans un langage on doit en plus se préoccuper des problèmes de syntaxe, ou de type d'instructions, propre à ce langage.

Notion de variable (1/2)

- 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
- Elle est une **boîte** que l'ordinateur va repérer par une **étiquette**.
- Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette

Notion de variable (2/2)

- La déclaration de variable consiste à créer la boîte et de lui coller une étiquette
- 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 (1/2)

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: A1 **exemple invalide: 1A**
 - doit être constitué uniquement de lettres, de chiffres et du soulignement _ (Eviter les caractères de ponctuation et les espaces)
valides: SMIP2007, SMP_2007 **invalides: SMP 2005,SMI-2007,SMP;2007**
 - 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 (2/2)

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

exemples: TotalVentes2004, Prix_TTC, Prix_HT

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 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← 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: $i \leftarrow 1$ $j \leftarrow i$ $k \leftarrow i+j$
 $x \leftarrow 10.3$ $OK \leftarrow FAUX$ $ch1 \leftarrow "SMI"$
 $ch2 \leftarrow ch1$ $x \leftarrow 4$ $x \leftarrow j$

(voir la déclaration des variables dans le transparent précédent)

- non valides: $i \leftarrow 10.3$ $OK \leftarrow "SMI"$ $j \leftarrow x$

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 (1/3)

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

Variables A, B, C: Entier

Début

A \leftarrow 3

B \leftarrow 7

A \leftarrow B

B \leftarrow A+5

C \leftarrow A + B

C \leftarrow B – A

Fin

Exercices simples sur l'affectation (2/3)

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 ?

Exercices simples sur l'affectation (3/3)

Ecrire un algorithme permettant d'échanger les
valeurs de deux variables 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*2, 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: =, <> (différent), <, >, <=, >=
 - 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:** $2 + 3 * 7$ vaut 23
- exemple:** $(2 + 3) * 7$ vaut 35

Les instructions d'entrées-sorties: lecture et écriture (1/2)

- Les instructions de lecture et d'écriture permettent à la machine de communiquer avec l'utilisateur
- La **lecture** permet d'entrer des données 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 (2/2)

- L'**écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
 - En pseudo-code, on note: **Ecrire (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

cette ligne est facultative en pseudo-code

variables A, B : entier

Début

écrire("entrer le nombre ")

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

Partie 2

***Les Tests
(Structures alternatives)***

Définition (1/3)

- La **structure Test**, encore appelée **structure alternative** permet de donner à l'ordinateur des séries d'instructions à effectuer selon que la situation se présente d'une manière ou d'une autre.
- Les instructions à effectuer dépendent de la **condition** posée
- Une condition est une comparaison entre deux valeurs à l'aide d'un opérateur de comparaison

Exemple de condition: $(5 < 2)$

5 et 2 représentent les valeurs

< représente l'opérateur de comparaison

Définition (2/3)

- Les valeurs peuvent être à priori de n'importe quel type (numérique, caractères...). Mais pour que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type
- L'ensemble des trois éléments composant la condition constitue une affirmation, qui à un moment donné est VRAIE ou FAUSSE.

Définition (3/3)

- Les opérateurs de comparaison sont:
 - Égal à (=)
 - Différent de (<>)
 - Strictement plus petit que (<)
 - Strictement plus grand que (>)
 - Plus petit ou égal à (<=)
 - Plus grand ou égal à (>=)

Les opérateurs de comparaison peuvent tout à fait s'employer avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique, les majuscules étant systématiquement placées avant les minuscules.

‘t’ < ‘w’

VRAI

“Maman” > “Papa”

FAUX

“maman” > “Papa”

VRAI

Syntaxe des Tests (1/2)

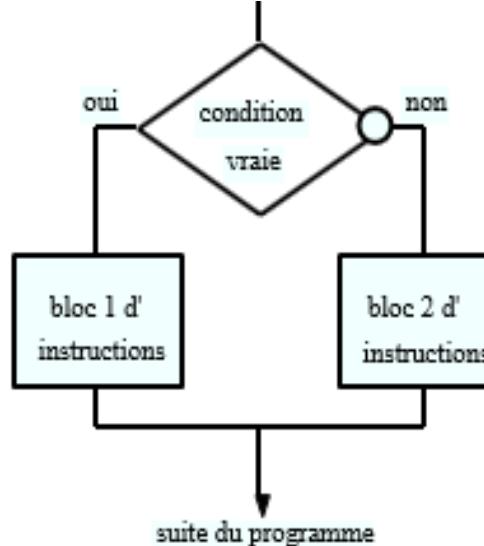
- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée
- On utilisera la forme suivante: **Si** condition **alors**

bloc d'instructions1

Sinon

bloc d'instructions2

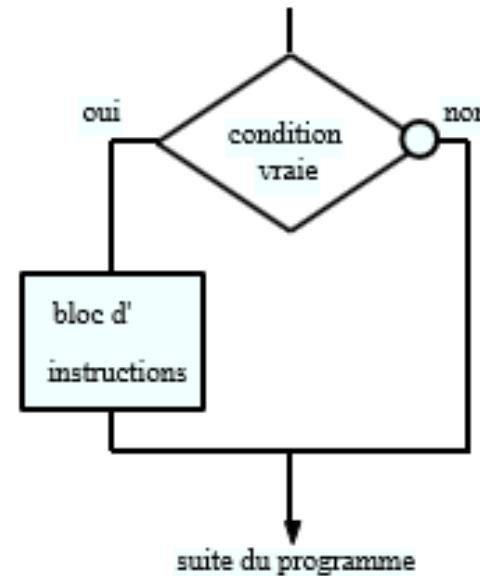
Finsi



Syntaxe des Tests (2/2)

- La partie Sinon n'est pas obligatoire quand aucun traitement n'est réalisé lorsque la condition est fausse.
 - On utilisera dans ce cas la forme simplifiée suivante:

Si condition **alors**
 bloc d'instructions
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

Exercice (tests)

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le nombre vaut zéro)

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

Exercice (condition composée)

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul).

Attention: toutefois on ne doit pas calculer le produit des deux nombres.

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

Tests imbriqués: exercice

Le prix de photocopies dans une reprographie varie selon le nombre demandé: 15 F la copie pour un nombre de copies inférieur à 10, 10 F pour un nombre compris entre 10 et 20 et 5 F 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*15

Sinon

Si (copies < 20) **Alors**

 prix ← copies*10

Sinon

 prix ← copies*5

Finsi

Finsi

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

Fin

Dans le cas de tests imbriqués, le **Sinon** et le **Si** peuvent être fusionnés en un **SinonSi**. On considère alors qu'il s'agit d'un seul bloc de test, conclu par un seul **FinSi**

Tests avec variable booléenne

Variables copies : entier

 prix : réel

 A,B: booléen

Début

 Ecrire ("Nombre de photocopies : ")

 Lire (copies)

 A \leftarrow copies < 10

 B \leftarrow copies < 20

Si A Alors

 prix \leftarrow copies*15

Sinon Si B Alors

 prix \leftarrow copies*10

Sinon

 prix \leftarrow copies*5

Finsi

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

Fin

Ici nous avons entré les conditions (copies < 10) et (copies < 20) dans les variables booléennes A et B respectivement. On teste maintenant la valeur de ses variables qui est soit Vraie, soit Fausse

Exercice

Proposer un algorithme qui donne l'état de l'eau selon sa température.

On rappelle que les trois états possibles sont: liquide, solide ou gazeuse

Sélection choix multiple

Si le traitement à faire dépend d'une valeur parmi tant d'autres, il est conseillé d'utiliser la sélection choix multiple:

Cas où variable vaut

valeur1: instruction1

valeur2: instruction2

.....

.....

valeurn: instructionn

autre: instruction par défaut

Fincas

Exercice choix multiple

Proposer un algorithme qui simule une mini-calculatrice effectuant les quatre opérations de base: addition , soustraction, multiplication et division.

L'algorithme fera saisir les opérandes et l'opérateur par l'utilisateur puis calcule et affiche le résultat

Partie 3

Les Structures répétitives

Définition

- Une structure répétitive (ou structure itérative) répète l'exécution d'un traitement dans un ordre précis, un nombre déterminé ou indéterminé de fois.
- Deux cas sont cependant à envisager, selon que:
 - Le nombre de répétitions est connu à l'avance: c'est le cas des boucles itératives
 - Le nombre de répétitions n'est pas connu ou est variable: c'est le cas des boucles conditionnelles

Utilité d'une structure répétitive

- Prenons le cas d'une saisie au clavier (une lecture), où par exemple, le programme pose une question à laquelle l'utilisateur doit répondre par **O** (Oui) ou **N** (Non).
- Tôt ou tard, l'utilisateur, maladroit, risque de taper autre chose que la réponse attendue
- Le programme peut alors planter, soit par:
 - Une erreur d'exécution: (**parce que le type de réponse ne correspond pas au type de la variable attendu**)
 - Une erreur fonctionnelle: (**il se déroule normalement jusqu'au bout, mais en produisant des résultats fantaisiste**)
- Dans tout programme un tant soit peu sérieux, on met en place ce qu'on appelle un **contrôle de saisie**, afin de vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme.

Instructions itératives: les boucles

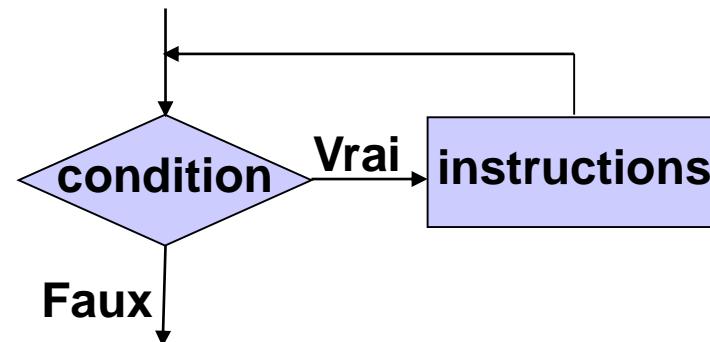
- Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois
- On distingue trois sortes de boucles en langages de programmation :
 - Les **boucles tant que** : on y répète des instructions tant qu'une certaine condition est réalisée
 - Les **boucles répéter** : on 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

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

⇒ **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

Boucle Tant que : exemple1

Contrôle de saisie d'une lettre majuscule jusqu'à ce que le caractère entré soit valable

Variable C : caractère

Debut

Ecrire (" Entrez une lettre majuscule ")

Lire (C)

TantQue (C < 'A' ou C > 'Z')

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

FinTantQue

Ecrire ("Saisie valable")

Fin

Boucle Tant que : exemple2

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

version 1

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 : exemple2 (version2)

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

version 2: attention à l'ordre des instructions et aux valeurs initiales

Variables som, i : entier

Debut

 som \leftarrow 0

 i \leftarrow 1

TantQue (som \leq 100)

 som \leftarrow som + i

 i \leftarrow i+1

FinTantQue

 Ecrire (" La valeur cherchée est N= ", i-1)

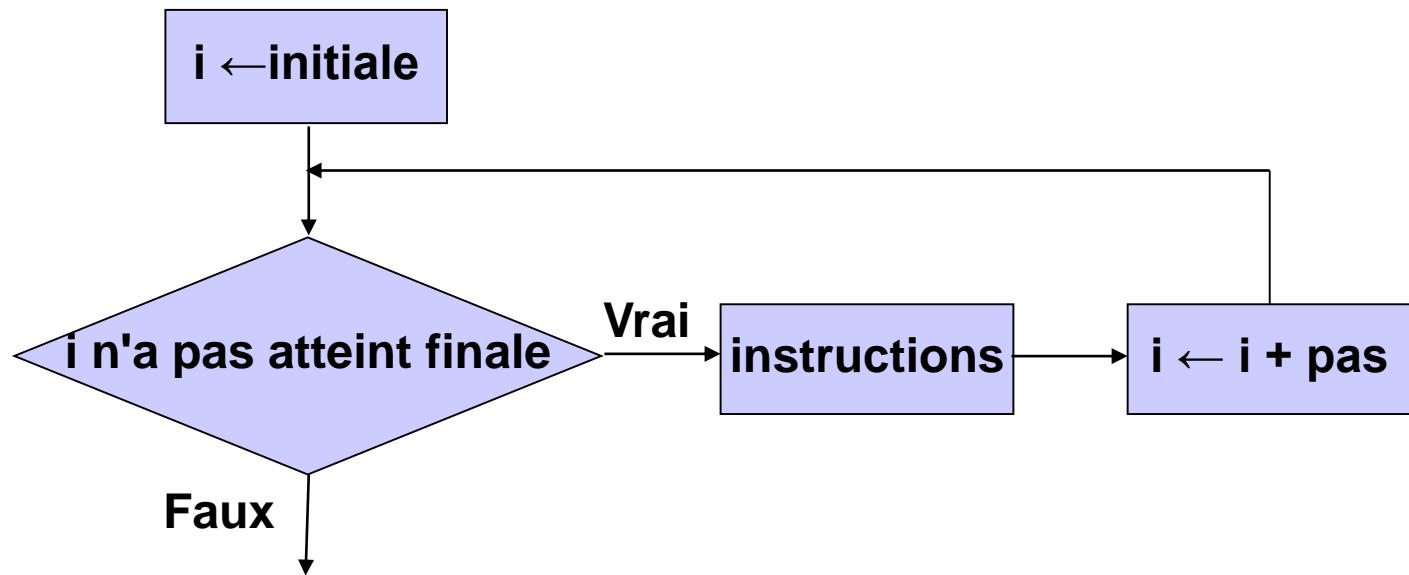
Fin

Les boucles Pour

Pour i allant de v_1 à v_2 par pas p

instructions

FinPour



Les boucles Pour

- Remarque : le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle
- Le compteur **i** est une variable de type entier (ou caractère). Elle doit être déclarée
- Le pas **p** est un entier qui peut être positif ou négatif. **p** 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
- **v1 et v2** 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 v1 est affectée à la variable compteur
- 2) On compare la valeur du compteur i et la valeur de finale v2 :
 - 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 : exemple1

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul

Variables x , puiss : réel
 n , i : entier

Debut

Ecrire (" Entrez la valeur de x ")

Lire (x)

Ecrire (" Entrez la valeur de n ")

Lire (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

Boucle Pour : exemple1 (version 2)

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul (**version 2 avec un pas négatif**)

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 **n à 1 par pas -1**

puiss \leftarrow puiss * x

FinPour

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

Fin

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^n où x est un réel non nul et n un entier positif ou nul (**version avec TantQue**)

Variables x , puiss : réel
 n, i : entier

Debut

Ecrire (" Entrez la valeur de x ")

Lire (x)

Ecrire (" Entrez la valeur de n ")

Lire (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

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

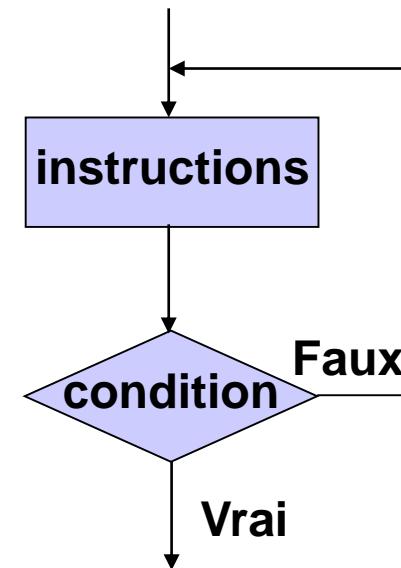
Exécution

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 vraie (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*
- Pour le choix entre **TantQue** et **Répéter** :
 - 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**

Partie 4

***Les Tableaux et les
Algorithmes de Tri***

Section 1

Les Tableaux

Exemple introductif

- Supposons que l'on veuille 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 **taille** (le nombre de ses éléments)
- La taille du tableau est prise selon les langages entre parenthèse ou entre crochets (avec parenthèses: **(taille)**, avec crochets: **[taille]**). C'est donc une affaire de convention. En pseudo-code, nous travaillerons avec les crochets.

Tableaux: Déclaration

- La **déclaration** d'un tableau a la structure suivante:
 - En pseudo code :
variable tableau identificateur[taille] : 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 (1)

- La déclaration précédente signifie que l'on demande une réservation de **taille** places en mémoire pour ranger les éléments du tableau
- **taille** est nécessairement une valeur numérique. Ca ne peut être en aucun cas une combinaison des variables du programme.
- 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
- Un grand avantage des tableaux est qu'on peut traiter les données qui y sont stockées de façon simple en utilisant des boucles

Tableaux : remarques (2)

La valeur d'un indice doit toujours:

- être égale au moins à 0 (dans quelques rares langages, le premier indice d'un tableau porte l'indice 1).
- être un nombre entier quel que soit le langage
- être inférieure ou égale au nombre d'éléments du tableau
 - Si l'indice commence par **0**, le plus grand indice du tableau a la valeur: **nombre d'éléments du tableau – 1**
 - Si l'indice commence par **1**, le plus grand indice du tableau a la même valeur que **nombre d'éléments du tableau**

Tout indice qui n'obéit pas les règles ci-dessus, déclenchera automatiquement une erreur

Tableaux : saisie

- Pour saisir un tableau, il faut parcourir tout le tableau et **lire** à chaque fois l'élément d'indice **i** du tableau :

variables i: entier

tableau T[10]: réel

Début

Pour i allant de 0 à 9

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

 lire (T[i])

FinPour

Fin

Tableaux : affichage

- Pour afficher un tableau, il faut parcourir tout le tableau et **écrire** à chaque fois l'élément d'indice **i** du tableau

variables i: entier

tableau T[10]: réel

Début

 écrire ("Les éléments du tableau ")

Pour i allant de 0 à 9

 écrire ("T[",i, "] =", T[i])

FinPour

Fin

Tableaux : exemple

- 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 : exercices

- Ecrire un algorithme qui déclare et remplit un tableau de 7 valeurs réelles en les mettant toutes à zéro
- Ecrire un algorithme qui déclare et remplit un tableau contenant les 6 voyelles de l'alphabet latin
- Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur

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, nous déclarerons un tableau à deux dimensions comme suit :

variable **tableau** identificateur**[taille1,taille2]** : 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

variables i,j : entier

tableau A[3,4]: réel

Début

Pour i allant de 0 à 2

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

Pour j allant de 0 à 3

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

 lire (A[i,j])

FinPour

FinPour

Fin

Le **2** de la première boucle (le **3** de la seconde) représente le **nombre de lignes-1**(le **nombre de colonnes-1**)

Tableau à deux dimensions : exercice

Ecrire un algorithme qui permet à l'utilisateur de saisir les éléments d'un tableau à deux dimensions de 5 lignes et de 7 colonnes. Les éléments du tableau sont des réels. Une fois la saisie terminée, l'algorithme affichera les éléments du tableau à l'écran

Section 2

Les Algorithmes de Tri

Tableaux : 2 problèmes classiques

- **Tri d'un tableau**
 - Tri par sélection
 - Tri à bulles
- **Recherche d'un élément dans un tableau**
 - Recherche séquentielle
 - 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 à bulles
 - ...
- Nous verrons dans la suite l'algorithme de tri par sélection et l'algorithme de tri à bulles. 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

- Nous pourrions décrire le processus de la manière suivante:
 - **Boucle principale:** prenons comme point de départ le premier élément puis le second, etc., jusqu'à l'avant dernier.
 - **Boucle secondaire:** à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel est le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

Tri par sélection : algorithme

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

#boucle principale : le point de départ se décale à chaque tour

Pour i allant de 0 à N-2

#on considère provisoirement que T[i] est le plus petit élément

 indice_ppe ← i

#on examine tous les éléments suivants

Pour j allant de i + 1 à N-1

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

 indice_ppe ← j

Finsi

FinPour

#A cet endroit, on sait maintenant où est le plus petit élément. Il ne reste plus qu'à effectuer la permutation.

 temp ← T[indice_ppe]

 T[indice_ppe] ← T[i]

 T[i] ← temp

#On a placé correctement l'élément numéro i, on passe à présent au suivant.

FinPour

Tri à bulle

- Le tri à bulle consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel tout élément est plus petit que celui qui le suit.
- **Principe de base du tri à bulle :**
 - Parcourir le tableau en comparant deux éléments consécutifs et les permutez s'ils sont mal placés
 - Deux éléments consécutifs sont mal placés si l'élément placé avant est supérieur à l'élément placé après

Tri à bulles : algorithme

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

$d \leftarrow N-2$

TantQue($d >= 0$)

Pour i allant de 0 à d

Si $T[i] > T[i+1]$ **alors**

 temp $\leftarrow T[i]$

$T[i] \leftarrow T[i+1]$

$T[i+1] \leftarrow temp$

Finsi

FinPour

$d \leftarrow d-1$

FinTantQue

Tri à bulles : exemple d'exécution

- Supposons que le tableau à trier par ordre croissant est le suivant:

9	4	1	7	3
---	---	---	---	---

Recherche séquentielle

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

Variables i: entier, Trouve : booléen

...

i \leftarrow 0 , Trouve \leftarrow Faux

TantQue ((i < N) ET (Trouve=Faux))

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

 Trouve \leftarrow Vrai

Sinon

 i \leftarrow i+1

FinSi

FinTantQue

Si Trouve **alors** # c'est équivalent à **Si Trouve=Vrai alors**

 écrire ("x appartient au tableau")

Sinon

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

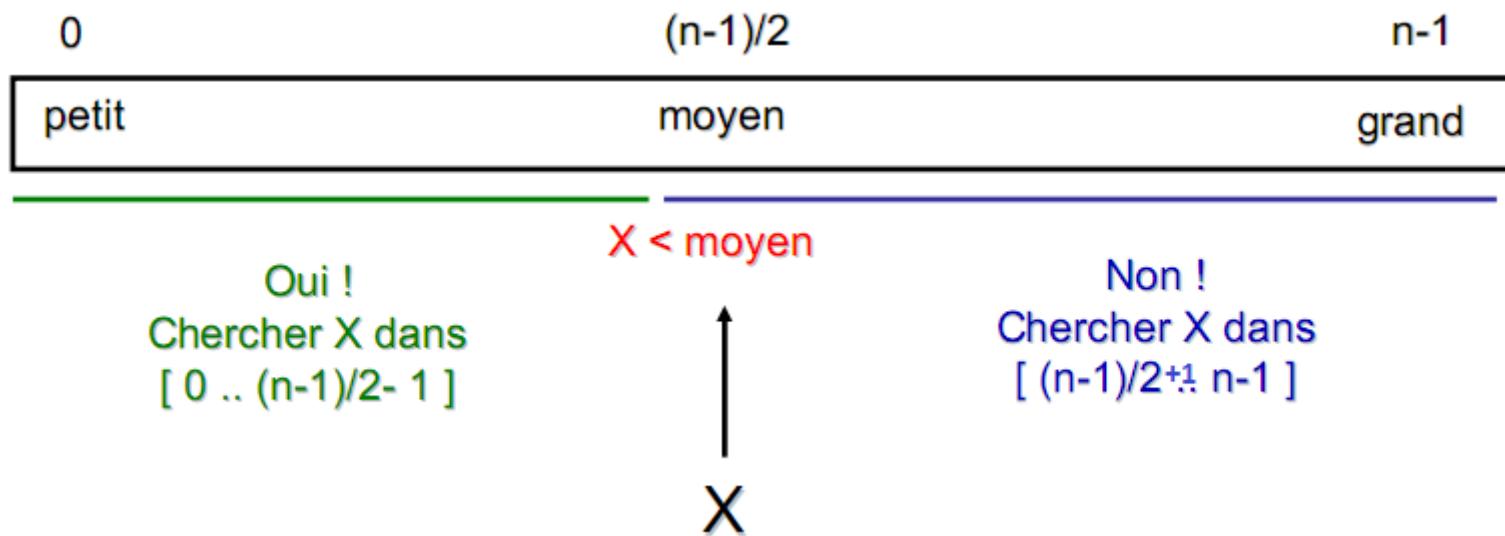
FinSi

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

- On utilise l'ordre pour
 - anticiper l'abandon dans une recherche linéaire,
 - guider la recherche: recherche dichotomie.



Recherche dichotomique : algorithme

inf←0 , sup←N-1, Trouve ← Faux

TantQue ((inf <=sup) ET (Trouve=Faux))

 milieu←(inf+sup) DIV 2

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

 Trouve ← Vrai

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

 inf←milieu+1

Sinon

 sup←milieu-1

FinSi

FinTantQue

Si Trouve **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

Partie 5

***Les Fonctions Prédéfinies/
Procédures et Fonctions***

Section 1

Les Fonctions Prédéfinies

Introduction

- Tout langage de programmation propose un certain nombre de fonctions;
 - Certaines sont indispensables car elles permettent d'effectuer des traitements qui seraient sans elles impossibles
 - D'autres servent à soulager le programmeur, en lui épargnant de longs et pénibles algorithmes

Structure des fonctions

- Une **fonction** est constituée de trois parties
 - Le nom de la fonction. Ce nom ne s'invente pas. Il doit impérativement correspondre à une fonction proposée par le langage
 - Deux parenthèses, une ouvrante, une fermante. Ces parenthèses sont toujours obligatoires, même s'il y a rien à l'intérieur.
 - Une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des arguments, ou des paramètres. Les arguments doivent être d'un certain type, et qu'il faut respecter ces types.

Exemple: **Sin(35)**

Nom de la fonction : **Sin**.

Argument : 35° à l'intérieur des parenthèses

Exercice d'application

- Parmi ces affectations, lesquelles provoqueront des erreurs, et pourquoi ?

Variables A, B, C : réel
D : Caractère

Début

```
A ← Sin(B)
A ← Sin(A + B * C)
B ← Sin(A) – Sin(D)
C ← Sin(A / B)
C ← Cos(Sin(A))
```

Fin

Les fonctions de texte

- Tous les langages proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre
 - **Len(c)** : renvoie le nombre de caractères de la chaîne **c**
 - **Mid(c,n1,n2)** : renvoie un extrait de la chaîne **c**, commençant au caractère numéro **n1** et faisant **n2** caractères de long.
 - **Left(c,n)** : renvoie les **n** caractères les plus à gauche de la chaîne **c**.
 - **Right(c,n)** : renvoie les **n** caractères les plus à droite de la chaîne **c**
 - **Trouve(c1,c2)** : renvoie la position de la chaîne **c2** dans la chaîne **c1**. Si **c2** n'est pas comprise dans **c1**, la fonction renvoie zéro.
- **Len**, **Mid**, **Left**, **Rigth** et **Trouve** sont respectivement les fonctions **NBCAR**, **STXT**, **GAUCHE**, **DROITE** et **CHERCHE** du logiciel ExecAlgo.

Les fonctions de texte : exemple

Application des fonctions	Résultats
Len ("Bonjour, ça va ?")	16
Len ("")	0
Mid ("Zorro is back", 4, 7)	"ro is b"
Mid ("Zorro is back", 12, 1)	"c"
Left ("Et pourtant...", 8)	"Et pourt"
Right ("Et pourtant...", 4)	"t..."
Trouve ("Un pur bonheur", "pur")	4
Trouve ("Un pur bonheur", "techno")	0

Les fonctions de texte : exercice

Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces

Les fonctions numériques

- **Ent(nbre)** : renvoie la partie entière d'un nombre
 - Exemple **Ent(3,228)** vaut **3**
- **Mod(n1,n2)** : renvoie le reste de la division du nombre n1 par n2.
 - Exemple **Mod(10,3)** vaut **1** car $10=3*3+1$
- **Alea()** : renvoie un nombre compris dans l'intervalle $[0;1[$.
 - Exemple Toto \leftarrow **Alea()** On a : $0 \leqslant \text{Toto} < 1$
 - si je veux générer un nombre entre 1,35 et 1,65 ; la « fourchette » mesure **1,65-1,35=0,30** de large. Donc $0 \leqslant \text{Alea()}*0,30 < 0,30$. Il suffit dès lors d'ajouter 1,35 pour obtenir la fourchette voulue.
$$\text{Toto} \leftarrow \text{Alea()}*0,30 + 1,35$$
- Ent, Mod et Alea sont respectivement les fonctions E (ou arrondi), reste et Rhasard du logiciel ExecAlgo.

Les fonctions de conversion

- Tous les langages proposent des fonctions destinées à opérer des conversions entre les chaînes de caractères et les numériques
 - **Cnum(c1)** : convertit la chaîne **c1** en numérique
 - Exemple **Cnum("3")** renvoie le nombre **3**
 - **Ccar(n)** : convertit le nombre **n** en chaîne de caractères
 - Exemple **Ccar(52)** renvoie la chaîne de caractère "**52**"

Les fonctions de conversion : exercice

Ecrivez un algorithme qui demande un entier positif à l'utilisateur et qui affiche sa conversion binaire.

Section 2

Les Procédures et Fonctions

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**, c'ad 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 (renvoyer)...

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 Facto suivante calcule le factoriel d'un entier x :

```
Fonction Facto (x : entier) : entier
    variables fact,i : entier
    fact ← 1
    Pour i ← 2 à x
        fact ← fact*i
    FinPour
    retourne (fact)
FinFonction
```

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

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

Utilisation des fonctions

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

variables z : entier, b : booléen

Début

b ← Pair(3)

z ← 5 * Facto(7)

écrire(" Facto(10)= ", Facto(10))

Fin

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

Fonctions : exercice

- Ecrire une fonction qui teste si deux réels passés en paramètre sont de même signe. La fonction renvoie la valeur **VRAI** s'ils sont de même signe et **FAUX** sinon. Utiliser la fonction définie dans un programme principal.

Procédures

- Dans certains cas, on peut avoir besoin de répéter une tâche dans plusieurs endroits du programme, mais que dans cette tâche 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 principal ou dans une autre procédure par une instruction indiquant le nom de la procédure :

Procédure exemple_proc (...)

...

FinProcédure

Exemple d'Appel de Procédure

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

Procédures : exercice

- Ecrire une procédure qui affiche à l'écran le factoriel de la longueur d'une chaîne de caractères passée en paramètre

Paramètres d'une procédure

- Les paramètres servent à échanger des données entre le programme principal (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

Transmission des paramètres (1)

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



- Notons bien que la flèche va seulement du paramètre réel vers le paramètre formel: le paramètre réel ne peut être modifié par l'appel de la procédure
 - **Remarque** : par défaut la transmission de paramètres se fait par valeur. Cette valeur peut être, d'une façon générale, celle d'une expression.

Transmission des paramètres (2)

- **La transmission par adresse (ou par référence, ou par variable)** : lorsque l'on désire modifier, ou calculer puis retourner la valeur d'un paramètre, celui-ci doit être passé par variable. 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 (3)

- Dans une procédure et/ou une fonction, un paramètre passé par valeur ne peut être qu'un paramètre en entrée.
- Le paramètre passé par adresse est un pointeur. A partir du moment où une variable est considérée comme un pointeur, toute affectation de cette variable se traduit automatiquement par la modification de la variable sur laquelle elle pointe.
- Passer un paramètre par référence, permet d'utiliser ce paramètre tant en lecture (en entrée) qu'en écriture (en sortie).

	passage par valeur	passage par référence
utilisation en entrée	oui	oui
utilisation en sortie	non	oui

Transmission des paramètres : exemples

Procédure incrementer1 (**x** : entier **par valeur**, **y** : entier **par adresse**)

$x \leftarrow x+1$

$y \leftarrow y+1$

FinProcédure

variables n, m : entier

Début

$n \leftarrow 3$

$m \leftarrow 3$

incrementer1(n, m)

écrire (" n= ", n, " et m= ", m)

Fin

résultat :

n=3 et m=4

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 **par valeur**, som, prod : entier **par adresse**)

```
som ← x+y  
prod ← x*y
```

FinProcédure

Procédure qui calcule la somme de 2 tableaux dans un troisième tableau :

Procédure SomTab (Tableau A,B : réel **par valeur**, Tableau C : réel **par adresse**, n: entier)

variable i : entier

```
Pour i← 0 à n-1  
    C[i]← A[i]+B[i]  
FinPour
```

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 principal
- **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

Récursivité

- Un module (fonction ou procédure) peut s'appeler lui-même: on dit que c'est un module **récursif**
- Tout module récursif doit posséder un cas limite (cas trivial) qui arrête la récursivité
- Exemple : Calcul du factoriel

```
Fonction facto (n : entier ) : entier
          Si (n=0) alors
                  retourne (1)
          Sinon
                  retourne (n*facto(n-1))
          Finsi
FinFonction
```

Les fonctions récursives

- Le processus récursif remplace en quelque sorte la boucle, c'est-à-dire un processus itératif.
- Il est à noter que l'on traite le problème à l'envers: on part du nombre, et on remonte à rebours jusqu'à 1 pour calculer le factoriel par exemple.
- Cet effet de rebours est caractéristique de la programmation récursive.

Les fonctions récursives : remarques

- La programmation récursive, pour traiter certains problèmes, peut être très économique, elle permet de faire les choses correctement, en très peu de lignes de programmation.
- En revanche, elle est très couteuse de ressources machine. Car il faut créer autant de variables temporaires que de « tours » de fonction en attente.
- Toute fonction récursive peut également être formulée en termes itératifs. Donc, si elles facilitent la vie du programmeur, elles ne sont pas indispensables.

Fonctions récursives : exercice (1)

- Ecrivez une fonction récursive (puis itérative) qui calcule le terme n de la suite de Fibonacci définie par :

$$U(0)=U(1)=1$$

$$U(n)=U(n-1)+U(n-2)$$

Fonction Fib (n : entier) : entier

Variable res : entier

Si (n=1 OU n=0) **alors**

res \leftarrow 1

Sinon

res \leftarrow Fib(n-1)+Fib(n-2)

Finsi

retourne (res)

FinFonction

Fonctions récursives : exercice (2)

- Une fonction itérative pour le calcul de la suite de Fibonacci :

Fonction Fib (n : entier) : entier

Variables i, AvantDernier, Dernier, Nouveau : entier

Si (n=1 OU n=0) **alors**

Nouveau \leftarrow 1

Finsi

AvantDernier \leftarrow 1, Dernier \leftarrow 1

Pour i \leftarrow 2 à n

Nouveau \leftarrow Dernier + AvantDernier

AvantDernier \leftarrow Dernier

Dernier \leftarrow Nouveau

FinPour

retourne (Nouveau)

FinFonction

NB: la solution récursive est plus facile à écrire

Procédures récursives : exemple

- Une procédure récursive qui permet d'afficher la valeur binaire d'un entier n

Procédure binaire (n : entier)

Si (n<>0) **alors**

 binaire (**Ent**(n/2))

 écrire (**Mod**(n,2))

Finsi

FinProcédure

Partie 6

***Les Variables Structurées et
Les Fichiers***

Section 1

Les Variables Structurées

Introduction (1/2)

- Nous pouvons réserver un emplacement (**variable**) pour une information d'un certain type
- Nous pouvons aussi réserver une série d'emplacements numérotés (**Tableau**) pour une série d'informations de **même type**
- Nous réservons maintenant une **série d'emplacements pour des données de types différents**
- Un tel emplacement s'appelle une **variable structurée**

Introduction (2/2)

- Lorsque nous utilisions des variables de type prédéfini, comme des entiers, des booléens, etc. nous n'avions qu'une seule opération à effectuer : **déclarer la variable en utilisant un des types existants**
- A présent que nous voulons créer un **nouveau type** de variable (par assemblage de types existants), il va falloir faire deux choses :
 - d'abord, créer le type
 - Ensuite, déclarer la (les) variable(s) d'après ce type.

Syntaxe

- **Création du type**

Structure *Nom_Structure*

 champ1: **Type1**

 champ2: **Type2**

 ...

 champn: **Typen**

FinStructure

exemple

Structure *Personne*

 Nom: **chaine de caractères**

 Prenom: **chaine de caractères**

 age: **entier**

 sexe: **caractère**

FinStructure

- **Déclaration de variable**

Variable *Nom_Variable*: **Nom_Structure**

Exemple

Variable *Individu*: **Personne**

Accès aux champs

- On accède aux différents champs d'une structure grâce au point « . »
- Soit par exemple la structure:

Structure *Personne*

Nom: **chaine de caractères**

Prenom: **chaine de caractères**

age: **entier**

sexe: **caractère**

FinStructure

Variable *Individu*: **Personne**

- **Individu.Nom** permet d'accéder au champ *Nom* de la variable structurée *Individu*
- **Individu.age** permet d'accéder au champ *age* de la variable structurée *Individu*

Remarques

- La seule opération globale sur un enregistrement est: recopier le contenu de r1 dans r2 en écrivant : $r1 \leftarrow r2$
 - *Ceci est équivalent (et plus efficace) que de copier champ à champ ; en plus on ne risque pas d'oublier un champ.*
 - *Il y a une condition : les 2 variables doivent être exactement du même type.*
- Lorsqu'on crée un type T2 à partir d'un type T1, ce type T1 doit déjà exister: donc T1 doit être déclaré avant T2
- Ce type permet de structurer proprement des informations qui vont ensemble, de les recopier facilement et de les passer en paramètres à des procédures

Exercice d'application

- Soit un étudiant caractérisé par son nom, son prénom, son âge et son numéro d'inscription :
 - *Proposer une procédure permettant de remplir les champs d'une variable de ce type en paramètre.*
 - *Proposer une procédure qui remplit un tableau de 10 étudiants. Le tableau est un paramètre de la procédure*
 - *Proposer une procédure qui affiche uniquement les numéros d'inscription des étudiants*
 - *Utiliser les sous programmes dans un programme principal*

Section 2

Les Fichiers

Introduction (1/2)

- Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources :
 - soit elles étaient incluses dans l'algorithme lui-même, par le programmeur;
 - soit elles étaient entrées en cours d'exécution par l'utilisateur.
- Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses.
- D'une exécution du programme à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme.

Introduction (2/2)

- Les fichiers sont là pour combler ce manque :
 - Ils servent à stocker des informations de manière permanente, entre deux exécutions d'un programme.
 - Ils sont stockés sur des périphériques à mémoire de masse (disquette, disque dur, CD Rom, etc.).

Organisation des fichiers

- Il existe deux catégories de fichiers :
 - Les fichiers texte
 - Les fichiers binaires.

Fichier texte

- C'est un fichier codé sous forme d'enregistrements (de lignes) càd que le fichier contient le même genre d'information à chaque ligne.
- Ce type de fichier est couramment utilisé dès lors que l'on doit stocker des informations pouvant être assimilées à une base de données.
- Toutes les données sont écrites sous forme de texte
 - **Exemple du carnet d'adresse**
le fichier stocke une personne (nom, prénom, téléphone et email) par ligne

Fichier binaire

- C'est un fichier qui ne possèdent pas de structure d'enregistrements.
- Tous les fichiers qui ne codent pas une base de données sont obligatoirement des fichiers binaires : **par exemple un fichier son, une image, un programme exécutable, etc.**
- Les données sont écrites à l'image exacte de leur codage en mémoire vive.
- Il est toujours possible d'opter pour une structure binaire même dans le cas où le fichier représente une base de données.

Le type d'accès

- le type d'accès désigne la manière dont la machine va pouvoir aller rechercher les informations contenues dans le fichier. On distingue :
 - **L'accès séquentiel** : on ne peut accéder qu'à la donnée suivant celle qu'on vient de lire. On ne peut donc accéder à une information qu'en ayant au préalable examiné celle qui la précède;
 - **L'accès direct (ou aléatoire)** : on peut accéder directement à l'enregistrement de son choix, en précisant le numéro de cet enregistrement;
 - **L'accès indexé** : il combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel. Il est particulièrement adapté au traitement des gros fichiers, comme les bases de données importantes.

Dans ce cours, on se limitera au type de base : **le fichier texte en accès séquentiel.**

Tableaux récapitulatif

	Fichiers Texte	Fichiers Binaires
On les utilise pour stocker...	des bases de données	tout, y compris des bases de données.
Ils sont structurés sous forme de...	lignes (enregistrements)	Ils n'ont pas de structure apparente. Ce sont des octets écrits à la suite les uns des autres.
Les données y sont écrites...	exclusivement en tant que caractères	comme en mémoire vive
Lisibilité	Le fichier est lisible clairement avec n'importe quel éditeur de texte	Le fichier a l'apparence d'une suite d'octets illisibles
Lecture du fichier	On ne peut lire le fichier que ligne par ligne	On peut lire les octets de son choix (y compris la totalité du fichier d'un coup)

Les instructions (1/3)

- Pour travailler sur un fichier, il faut d'abord l'ouvrir. Lorsqu'on ouvre un fichier, on précise ce qu'on va en faire : **lire**, **écrire** ou **ajouter**.
 - Si on ouvre un fichier **pour lecture**, on pourra uniquement récupérer les informations qu'il contient, sans les modifier en aucune manière.
 - **Ouvrir "Exemple.txt" pour Lecture**
 - Si on ouvre un fichier **pour écriture**, on pourra mettre dedans toutes les informations que l'on veut. Mais les informations précédentes seront intégralement écrasées
 - **Ouvrir "Exemple.txt" pour écriture**
 - Si on ouvre un fichier **pour ajout**, on ne peut ni lire, ni modifier les informations existantes. Mais on pourra ajouter de nouvelles lignes.
 - **Ouvrir "Exemple.txt" pour ajout**

Les instructions (2/3)

- L'instruction **LireFichier** *Nom_Fichier* permet de récupérer l'enregistrement suivant par rapport au dernier enregistrement lu dans le fichier *Nom_Fichier* dans une variable spécifiée

Ouvrir "Exemple.txt" pour Lecture

LireFichier "Exemple.txt", Truc

Cela permet de récupérer le premier enregistrement du fichier Exemple.txt, dans la variable Truc.

- L'instruction **EcrireFichier** *Nom_Fichier* permet d'ajouter la variable spécifiée dans le fichier

Ouvrir "Exemple.txt" pour écriture (ajout)

EcrireFichier "Exemple.txt", Truc

Cela permet d'ajouter la variable Truc dans le fichier Exemple.txt.

Les instructions (3/3)

- Pour lire un fichier séquentiel de bout en bout, il faut programmer une **boucle**. Pour cela, on utilise la fonction **EOF(*Nom_Fichier*)** (acronyme pour **End Of File**). Cette fonction renvoie la valeur **Vrai** si on a atteint la fin du fichier.
- Une fois qu'on en a terminé avec un fichier, il faut toujours le fermer grâce à l'instruction **Fermer *Nom_Fichier***

Variable Truc : Chaîne de caractères

Début

Ouvrir "Exemple.txt" pour Lecture

Tantque Non EOF("Exemple.txt")

LireFichier "Exemple.txt", Truc

Ecrire(Truc)

...

FinTantQue

Fermer "Exemple.txt"

Fin

Exercice d'application

- Soit un étudiant caractérisé par son nom, son prénom, son âge et son numéro d'inscription :
 - *Proposer une procédure permettant de remplir les champs d'une variable de ce type en paramètre.*
 - *Proposer une procédure qui remplit un tableau de 10 étudiants. Le tableau est un paramètre de la procédure*
 - *Créer et copier les éléments du tableau dans un fichier **BlocNote.txt***