Université Mohammed V
Faculté des Sciences
Département d'informatique

Cours de l'algorithmique et programmation: Licence SMI- S2 (Accréditation : 2014-2018)

Pr. Y. EL BENANI

Année de production : 2014

Chap1: L'introduction à l'algorithmique

Cours d'Algorithmique 1

1ère année SMI

Département d'Informatique,

Université Mohammed V

elbenani@hotmail.com

Objectif du cours

- **Objectifs :**
  - Apprendre les concepts de base de l'algorithmique.
  - S'initier à l'analyse et la résolution de problèmes et écrire les algorithmes correspondants.
  - Étudier les procédures et les fonctions qui permettent de structurer et de réutiliser les algorithmes.
  - Avoir une première notion de performance des algorithmes utilisés.

Plan du cours

Galerie de portraits

George BOOLE
(1815-1864)

Mathématicien anglais, il publie en 1854 les Lois de la pensée. Dans ce livre, il décrit comment toute la logique peut être définie par un principe simple: le binaire.

John Von NEUMANN
(1903-1957)

L'un des personnages clés des débuts de l'informatique. Il publia de nombreux articles sur l'algèbre et la mécanique quantique avant de se consacrer à la construction d'ordinateurs et à la modélisation mathématique de la réaction en chaîne de la bombe A. Ses "machines IAS" sont à l'origine de "l'Architecture Von NEUMANN", c'est à dire celle des ordinateurs tels que nous les connaissons.

Galerie de portraits

Grace Murray HOPPER
(1906 - 1992)

Cette américaine, mobilisée comme auxiliaire dans la marine américaine fut affectée aux travaux de programmation et d'exploitation de l'ENIAC. Puis, devenue une grande spécialise de la programmation des ordinateurs, elle sera l'une des principales créatrices du COBOL.

Alan TURING
(1912 - 1954)

Mathématicien anglais, maître-assistant à Cambridge dès 23 ans.
Il a conçu en 1936 une machine logique capable de résoudre tous les problèmes que l'on peut formuler en termes d'algorithmes.
Pendant la guerre, il participera à la réalisation de la *Bombe*, première machine électromécanique de décryptage des messages codés avec l'Enigma Allemande.

Galerie de portraits

Dennis RITCHIE
(1941)

Cet ingénieur des laboratoires Bell, est l'auteur du langage C. En 1973, avec K. THOMPSON, il réécrira dans ce nouveau langage le système d'exploitation UNIX.

Vinton G. CERF
(1943 -)

C'est l'un des pères de l'Internet. Encore étudiant de l'université de Los Angeles, il fut l'un des auteurs du protocole TCP/IP et développa avec une équipe de chercheurs les premiers outils utilisant ce mode de communication. Il est aujourd'hui président de *l'Internet Society* qui surveille les nouveaux standards d'Internet.

Galerie de portraits

Bjarne STROUSTRUP
(1950 -)

Créateur du langage C++ basé sur le langage C mais en lui donnant une dimension de Langage Orienté Objet.

James Gosling
(1955 -)

Créateur du langage Java basé sur le langage C++. La particularité principale de Java est que les logiciels écrits dans ce langage sont très facilement portables sur plusieurs systèmes d'exploitation.

Galerie de portraits

Bill GATES
(1951 -)

Ancien président (et fondateur avec P. ALLEN) de Microsoft. Cette société est à l'origine du MS-DOS, de Windows, du Basic-Microsoft puis de Visual Basic.

Steve JOBS
(1955 - 2011)

L'un des fondateurs de la société *Apple*.
Après son éviction d'Apple S. JOBS créera la société Next avant d'être rappelé pour redresser Apple.

Galerie de portraits

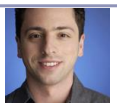
Richard STALLMAN
(1953 -)

Fondateur du projet GNU, lancé en 1984 pour développer le système d'exploitation libre GNU et donner ainsi aux utilisateurs des ordinateurs la liberté de coopérer et de contrôler les logiciels qu'ils utilisent. Il est également le créateur (entre autres) de l'éditeur Emacs et du compilateur gcc.

Linus TORVALDS
(1969 -)

Finlandais d'origine, il a construit en 1991 un nouveau système d'exploitation de type UNIX appelé Linux. Ayant choisi de le diffuser suivant le principe des logiciels libres, Linus TORVALDS ne retire aucune royaltie de son travail sur le noyau Linux.

Galerie de portraits

Larry Page (1973 -) Sergey Brin (1973 -)

Créateurs du moteur de recherche Google. Ces deux jeunes brillants nord-américains ont lancé leur moteur de recherche en 1999. Ce mot vient du terme "googol" qui désigne un chiffre, un 1 suivi de 100 zéros, traduisant l'exhaustivité du moteur de recherche.

Mark Zuckerberg (1984 -)

Créateur de Facebook
C'est en 2004 que la première version de Facebook voit le jour pour mettre en relation les étudiants de Harvard.

Pourquoi un cours d'algorithmique

- Pour proposer à la « machine» d'effectuer un travail à notre place.
- Problème : expliquer à la machine comment elle doit le faire.
- Besoins :
  - savoir expliquer et formaliser son problème
  - Concevoir et écrire des algorithmes (séquence d'instructions qui décrit comment résoudre un problème particulier).

Les algorithmes sont anciens !

- **Les algorithmes ne sont pas nés avec l'informatique** :
  - L'algorithme d'Euclide pour calculer le PGCD de deux entiers est vieux de plus de 2000 ans !
  - Des descriptions précises d'algorithmes sont présents dans la Chine ancienne.
    (Par exemple, pour extraire des racines carrées à partir de divisions effectuées sur une « surface à calculer »).

Les origines de l'algorithmique

- **Mohammed Al-Khwarizmi (780 - 850)**
- أبو عبد الله محمد بن موسى الخوارزمي ou ابوجعفر محمد بن موسى خوارزمى
- Mathématicien, géographe, astrologue et astronome musulman arabe dont les écrits ont permis l'introduction de l'algèbre en Europe.
- **L' origine du mot « algorithme »** est lié au nom d'Al-Khwarizmi.

  Ce savant arabe a publié plusieurs méthodes pour le calcul effectif de racines d'une équation du second degré et grâce à lui les chiffres arabes ont pu se diffuser en occident.

Algorithme

- Savoir expliquer comment faire un travail sans la moindre ambiguïté.
- Un algorithme : est une suite finie d'instructions que l'on applique à un nombre fini de données dans un ordre précis pour arriver à un résultat.
- L'écriture algorithmique : un travail de programmation ayant une vision universelle :
  - Un algorithme ne dépend pas du langage dans lequel il est implanté,
  - ni de la machine qui va exécuter le programme correspondant.

Algorithmique

- **L'algorithmique** désigne 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

Propriétés d'un algorithme

- Un algorithme doit:
  - avoir un **nombre fini d'étapes**,
  - avoir un **nombre fini d'opérations** par étape,
  - **se terminer** après un nombre fini d'opérations,
  - fournir un **résultat**.
- Chaque opération doit être:
  - **définie** rigoureusement et sans ambiguïté
  - **effective**, c.-à-d. réalisable par une machine
- Le comportement d'un algorithme est **déterministe**.

Les 3 étapes d'un algorithme

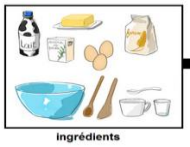
- Les entrées (les données du problème)
- Le traitement
- Les sorties (l'affichage des résultats)

- **Les entrées :** Il s'agit de repérer les données nécessaires à la résolution du problème.
- **Le traitement :** Il s'agit de déterminer toutes les étapes des traitements à faire et donc des "**instructions**" à développer pour arriver aux résultats.

Les 3 étapes d'un algorithme

- **Les sorties :** les résultats obtenus peuvent être affichés sur l'écran, ou imprimés sur papier, ou bien encore conservés dans un fichier.

Exemple d'algorithme : recette de cuisine



ingrédients

RECETTE (ALGORITHME)

crêpes

Mettre la farine dans une terrine

Former un puits

Mettre les œufs entiers, le sucre, l'huile et le beurre

Mélanger délicatement avec un fouet en ajoutant au fur et à mesure le lait.

Faire chauffer une poêle anti-adhésive et y déposer quelques gouttes d'huile

Faire cuire les crêpes à feu doux.

Exemple d'un algorithme

- On se donne deux points A et B du plan.
  - 1. Tracer le cercle de centre A passant par B.
  - 2. Tracer le cercle de centre B passant par A.
  - 3. Nommer C et D les points d'intersection de ces cercles.
- Construire le polygone ADBC.
- Cet algorithme décrit la construction d'un losange dont une diagonale est [AB].
- *Les entrées* sont : les points A et B.
- *Le traitement* de la construction est décrit dans les phases 1. 2. et 3.
- *La sortie* est : le polygone ADBC.

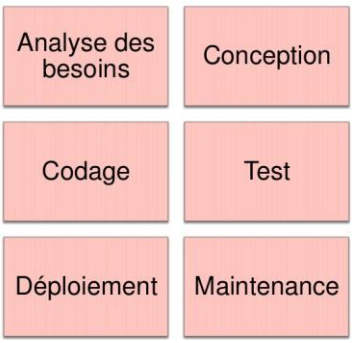
Algorithme de calcul

- Étape 1 : choisir un nombre : x
- Étape 2 : lui ajouter 4 : x+4
- Étape 3: multiplier la somme obtenue par le nombre choisi : (x+4)*x
- Étape 4 : ajouter 4 à ce produit : (x+4)*x+4
- Étape 5 : afficher le résultat : $(x+2)^2$

Génie logiciel

- Définition: le génie logiciel regroupe les sciences et les technologies qui permettent la production et la maintenance de logiciels de qualité
- Le cycle de vie d'un logiciel regroupe les étapes de production du logiciel, ainsi que leur ordonnancement.

Les étapes de développement du logiciel



Les étapes de développement du logiciel

- 1) Analyse des besoins : que fait le système ?
  - On définit les fonctionnalités du système à développer.
- 2) conception : comment faire le système ?
  - décomposition du système en modules logiciels et matériel.
- 3) Implémentation (codage) :
  - Réalisation des programmes dans un langage de programmation

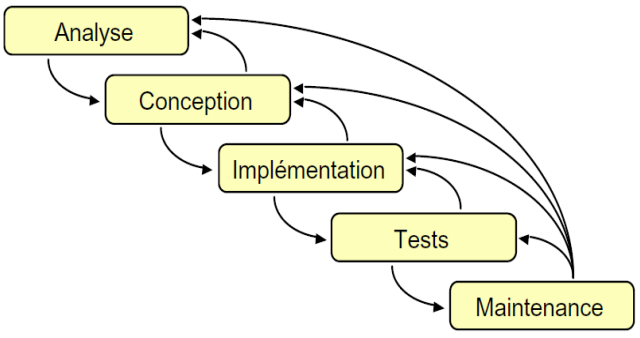
Les étapes de développement du logiciel

- 4) tests unitaires : effectuer les tests de chaque composant du logiciel en vue de son intégration.
- 5) intégration : Intégration des modules et test de tout le système
- 6) Livraison et maintenance:
  - Livraison du produit final à l'utilisateur,
  - Le suivi, les modifications, les améliorations après livraison.

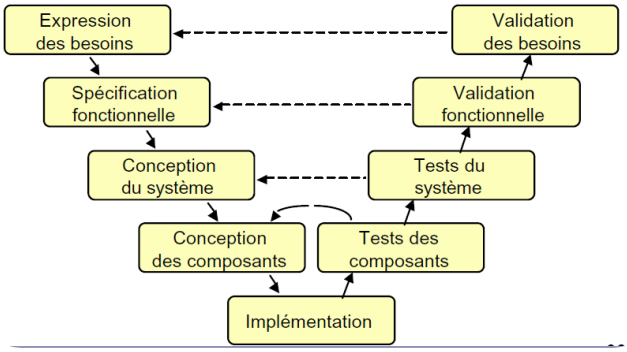
Modèles de développement

- Objectifs :
  - Organiser les différentes phases du cycle de vie pour l'obtention d'un logiciel fiable, adaptable et efficace.
  - Guider le développeur dans ses activités techniques
  - Fournir des moyens pour gérer le développement et la maintenance (ressources, délais, avancement, etc.).
- Il existe plusieurs types de modèles : en cascade, en V, en spirale…

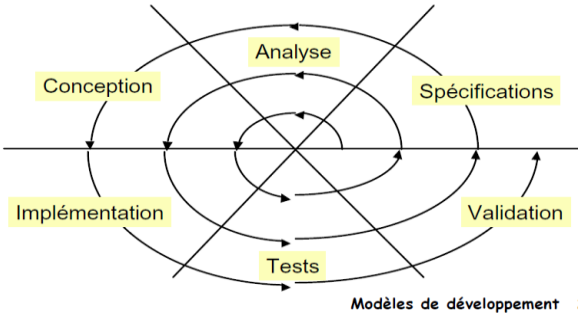
Modèle en cascade



Modèle en V



Modèle en spirale



Les langages de programmation

- Le langage de programmation est l'intermédiaire entre l'humain (anglais) et la machine (binaire).
- Il existe des milliers de langages de niveau élevé, pour tous les goûts et toutes les applications.
- Quelques uns des plus connus: C, C++, Java, PHP,…

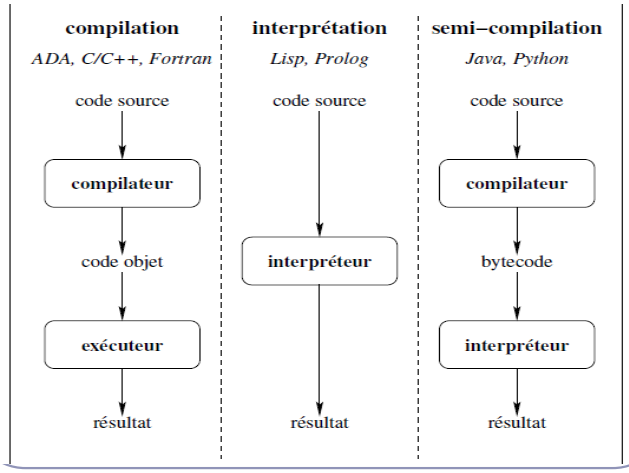
Les langages de programmation

- Haut niveau : proche de l'homme, vocabulaire et syntaxe plus riches
  - C++, Java, PHP,…
  - C, Fortran,…
  - Assembleur
  - Langage machine
- Bas niveau : proche de la machine, instructions élémentaires

Compilation et interpréteur

- **Compilation** : permet de traduire le code source du programme vers le langage natif (objet) de la machine (ou parfois vers du code intermédiaire).

- **Interpréteur** : permet de traduire et d'exécuter chaque instruction du programme. Ce mécanisme est utilisé pour le passage d'un programme précompilé à un pseudo-code (cas de Java).

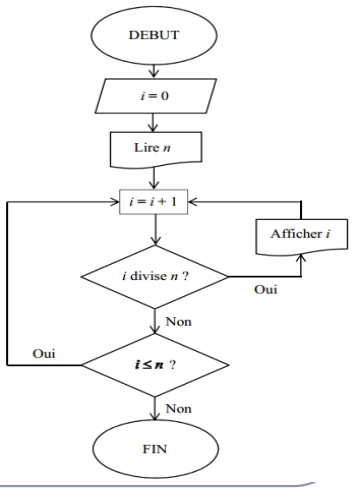


Représentation d'un algorithme

- Un algorithme, pour être lu par tous, est écrit en langage naturel (pseudo-code) ou représenté par un organigramme.
- Un programme est la traduction d'un algorithme pour être compris par une machine (ordinateur, calculatrice, téléphone,…).

Représentation d'un algorithme

- L'organigramme: représentation graphique avec des symboles (réctangles, losanges, etc.)
- offre une vue d'ensemble de l'algorithme.
- Représentation presque abandonnée aujourd'hui.



Représentation d'un algorithme

- Le pseudo-code: représentation textuelle avec une série de conventions ressemblant à un langage de programmation.
- plus pratique pour écrire un algorithme.
- représentation largement utilisée.
- Un algorithme écrit en pseudo-code est composé de trois parties suivantes : L'en-tête, la partie déclarative et le corps

```
Algorithme Surface_cercle ;          L'en-tête
Constante Pi = 3,14 ;
Variable  R, Surf : Réel ;           Les déclarations
Début
   Ecrire (' Donnez la valeur de rayon: ') ;
   Lire (R) ;
                                     Le corps
   Surf ⟵ Pi*R^2 ;
   Ecrire (' La surface de cercle est : ', Surf) ;
Fin.
```

Les instructions de base

- Un programme informatique est formé de quatre types d'instructions considérées comme des briques de base :
  - Les affectations de variables
  - Les lectures et/ou les écritures
  - Les tests
  - Les boucles

Formalisme d'un algorithme

- Un algorithme informatique doit suivre les règles suivantes :
  Il est composé d'une entête et d'un corps :
  - l'entête, qui spécifie :
    - le nom de l'algorithme (**Nom :)**
    - son utilité (**Rôle :)**
    - Les données "en entrée", c.-à-d. les éléments qui sont indispensables à son bon fonctionnement (**Entrée :)**

Formalisme d'un algorithme

    - les données "en sortie", c.-à-d. les éléments calculés, produits, par l'algorithme (**Sortie :)**
    - les données locales à l'algorithme et indispensables (**Déclaration :)**

- le corps, qui est composé :
  - du mot clé **début**
  - d'une suite d'instructions
  - du mot clé **fin**

Formalisme d'un algorithme : exemple

Nom : addDeuxEntiers
Rôle : Additionner deux entiers a et b et mettre le résultat dans c
Entrée : a, b deux entiers
Sortie : c un entier
Déclaration : a, b, c : entier
début
c←a+b;
ecrire ("la somme de a et b est =", c);
fin

Chap2: Notion de variables et d'affectation

Algorithmique

Notions et Instructions de base

Notion de variable

- Une **variable** sert à stocker la valeur d'une donnée dans un langage de programmation.
- Une variable désigne un emplacement mémoire (boites étiquetées) dont le contenu peut changer au cours d'un programme (d'où le nom de variable).
- Chaque emplacement mémoire a un numéro qui permet d'y faire référence de façon unique : c'est l'adresse mémoire de cette cellule.

Notion de variable

- Règle : La variable doit être **déclarée** avant d'être utilisée, elle doit être caractérisée par :
  - un nom (**Identificateur**).
  - un **type** qui indique l'ensemble des valeurs que peut prendre la variable (entier, réel, booléen, caractère, chaîne de caractères, …).
  - Une valeur.

Identificateurs : règles

Le choix du nom d'une variable est soumis à quelques règles qui varient selon le langage, mais en général:

- Un nom doit commencer par une lettre alphabétique
  exemple : E1 (1E n'est pas valide)
- doit être constitué uniquement de lettres, de chiffres et du soulignement (« _ ») (Éviter les caractères de ponctuation et les espaces).

  Exemples : SMI2009, SMI_2009

  SMP 2009, SMP-2009, SMP;2009 : sont non valides

Identificateurs : règles

- doit être différent des mots réservés du langage (par exemple en C : **int, float, double, switch, case, for, main, return**, …).
- La longueur du nom doit être inférieure à la taille maximale spécifiée par le langage utilisé.

Identificateurs : conseils

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

exemples: NoteEtudiant, Prix_TTC, Prix_HT

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

les variables

- Le type d'une variable détermine l'ensemble des valeurs qu'elle peut prendre.
- Toute variable utilisée dans un programme doit faire l'objet d'une déclaration préalable.
- En pseudo-code, la déclaration de variables est effectuée par la forme suivante :

  Variables id1, id2, … : type
- Cette instruction permet de réserver de l'espace mémoire pour les variables nommées id1, id2, …
- Dépendant du **type** de données : entiers, réels, caractères, etc.

les variables

- **Exemple de variables**

```
Variables   age, compteur : entier;
            note, moyen : réel;
                  OK: booléen;
            nom, prenom : chaîne de caractères;
```


Types des variables

- Les types offerts par la plus part des langages sont :
  - **Type numérique : entier ou réel**
    - **Byte** (codé sur 1octet): de $[-2^7,2^7[$ ou $[0, 2^8[$
    - **Entier court** (codé sur 2 octets) : $[-2^{15},2^{15}[$
    - **Entier long** (codé sur 4 octets): $[-2^{31},2^{31}[$
    - **Réel simple précision** (codé sur 4 octets) : précision d'ordre 10^{-7}
    - **Réel double précision** (codé sur 8 octets) : précision d'ordre 10^{-14}

Type numérique : entier ou réel

- **Operateurs arithmétiques :**
  - **+ (addition)**
  - **- (soustraction)**
  - *** (produit)**
  - **/ ou div (quotient)**
  - **% ou mod (reste de la division entière)**
- **Opérateurs de comparaison :**
  - **<, >, =, <=, >= : selon l'ordre alphabétique**

Types des variables

- **Type booléen**: deux valeurs VRAI ou FAUX
  - Opérateurs : ET, OU et NON
- **Type caractère:** lettres majuscules, minuscules, chiffres, symboles,.. **Exemples : 'A', 'b', '1', '?', …**
  - Opérateurs : **<, >, =, <=, >= , selon l'ordre alphabétique**
- **Type chaîne de caractère** (string en anglais) : toute suite de caractères. Le contenu de ces variables est noté entre guillemets (**" " : pour ne pas confondre 777 et "777"**);
- **Exemples**: " " , " Nom, Prénom", "code postale: 1000"

Variables : remarques

- Pour le type numérique, on va se limiter aux entiers et réels sans considérer les sous types
- Pour chaque type de variables, il existe un ensemble d'opérations correspondant.
- Une variable est l'association d'un nom avec un type, permettant de mémoriser une valeur de ce type.
- Une fois qu'un type de données est associé à une variable, cette variable ne peut plus en changer.
- Une fois qu'un type de données est associé à une variable le contenu de cette variable doit **obligatoirement être du même type.**

Constante

- Une constante est une variable dont la valeur ne change pas au cours de l'exécution du programme.
- En pseudo-code, **Constante identificateur ← valeur : type**
  (par convention, les noms de constantes sont en majuscules)
- Exemple : pour calculer la surface des cercles, la valeur de pi est une constante mais le rayon est une variable.
  Constante PI ← 3.14 : réel, MAXI ← 32 : entier
- Une constante doit toujours recevoir une valeur dès sa déclaration.

Affectation

- **L'affectation** consiste à attribuer une valeur à une variable (c'est-à-dire remplir ou modifier le contenu d'une zone mémoire).
- En pseudo-code, l'affectation est notée par 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.

Affectation

Exemples :

i ←1; j ←i; k ←i+j;
x ←10.3 ; val ← 132;
OK ←FAUX ;
ch1 ←"SMI";
ch2 ←ch1 ; x ←4; x ←j
delta ← b*b - 4*a*c;

variable contenant une valeur

(avec i, j, k : entier; x, a, b, c, delta :réel; ok :booléen; ch1,ch2 :chaine de caractères)

- Exemples **non valides:**
  i ←10.3 ;
  OK ←"SMI";
  j ←x;

Affectation

- Les langages de programmation C, C++, Java, … utilisent le signe égal **=** pour l'affectation ←.

  Remarques :
  - Lors d'une affectation, l'expression de droite est évaluée et la valeur trouvée est affectée à la variable de gauche.
  - Ainsi, A←B ; est différente de B←A;

Affectation : remarques

- l'affectation est différente d'une équation mathématique :
  - Les opérations x ← x+1 et x ← x-1 ont un sens en programmation et se nomment respectivement incrémentation et décrémentation.
  - A+1 ← 3 n'est pas possible en langages de programmation et n'est pas équivalente à A ← 2
- Certains langages donnent des valeurs par défaut aux variables déclarées.
- Pour éviter tout problème, il est préférable **d'initialiser les variables** déclarées.

Syntaxe générale de l'algorithme

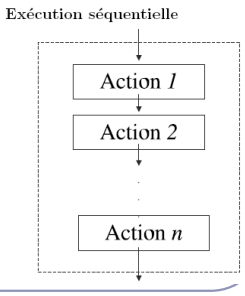
```
Algo exemple
    /* La partie déclaration de l'algorithme */
Constantes // les constantes nécessitent une valeur dès leur déclaration
    var1←20  : entier
    var2←"bonjour!"  : chaîne
Variables  // les variables proprement dites
    var3, var4  : réels
    var5  : chaîne
Début  // corps de l'algorithme

    /* instructions */

Fin
```

la séquence des instructions

- Les opérations d'un algorithme sont habituellement exécutées une à la suite de l'autre, en séquence (de haut en bas et de gauche à droite).
- L'ordre est important.
- On ne peut pas changer cette séquence de façon arbitraire.
- Par exemple, enfiler ses bas puis enfiler ses bottes n'est pas équivalent à enfiler ses bottes puis enfiler ses bas.



Affectation : exercices

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

Variables A, B, C: **Entier**
Début
A ← 7;
B ← 17;
A ← B;
B ← A+5;
C ← A + B;
C ← B – A;
Fin

A	B	C
7	-	-
7	17	-
17	17	-
17	22	-
17	22	39
17	22	5

Affection : exercices

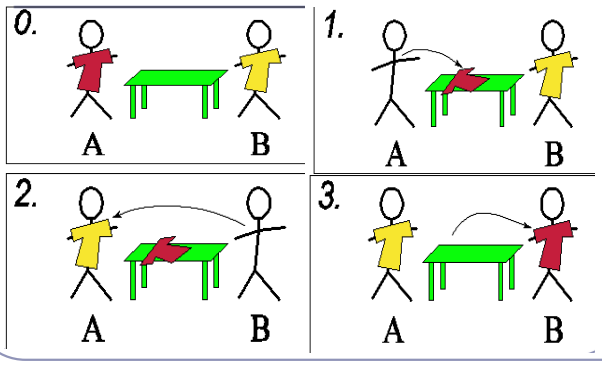
Donnez les valeurs des variables A et B après exécution des instructions suivantes ?

Variables A, B : **Entier**
Début
A ← 6;
B ← 2;
A ← B;
B ← A;
Fin

A	B
6	-
6	2
2	2
2	2

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

Affectation : l'échange des chandails



Affectation : échanges

Écrire un algorithme permettant d'échanger les valeurs de deux variables A et B ?

Réponse :

on utilise une variable auxiliaire C et on écrit les instructions suivantes :

C ← A ;
A ← B ;
B ← C ;

Opérateur

- Un opérateur est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs".
- Il peut être unaire ou binaire.

- Exemple :

non A (A variable de type booléen) : cas unaire

a + b : cas binaire (a et b entiers)

Les types d'opérateurs

- Les **opérateurs** dépendent du type de l'opération, ils peuvent être :
  - des opérateurs arithmétiques: +, -, *, /, % (modulo), ^(puissance)
  - des opérateurs logiques: NON(!), OU(| |), ET (&&)
  - des opérateurs relationnels: =, <, >, <=, >=
  - des opérateurs sur les chaînes: & (concaténation)

Les opérandes

- Une opérande est une entité (variable, constante ou expression) utilisée par un opérateur.

Exemple :

Variables a, b : entier; x, y : réel;

- a + b

a est l'opérande gauche et b est l'opérande droite

- a+ 200;
- x - y;

Les expressions

- 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.
- Une expression est évaluée de gauche à droite mais en tenant compte des **priorités** des opérateurs**.**

Priorité des opérateurs

- Pour les opérateurs arithmétiques donnés ci-dessous, l'ordre de priorité est le suivant (du plus prioritaire au moins prioritaire) :

  - **( ) : les parenthèses**
  - **^ : (élévation à la puissance)**
  - *** , /, % (multiplication, division, modulo)**
  - **+ , - (addition, soustraction)**
  - **<, <=, >, >=**
  - **==, !=**

Priorité des opérateurs

Exemple : 9 + 3 * 4 vaut 21 (et non 48)

- En cas de besoin, on utilise les parenthèses pour indiquer les opérations à effectuer en priorité

Exemple : (9 + 3) * 4 vaut 48

- À priorité égale, l'évaluation de l'expression se fait de gauche à droite.
- Exemple :

  15 / 5 * 3 = 9 et non 1

  5 – 2 + 4 = 7 et non –1

Expression : remarques

- On ne peut pas additionner un entier et un caractère
- Toutefois dans certains langages on peut utiliser un opérateur avec deux opérandes de types différents, c'est par exemple le cas avec les types arithmétiques (4 + 5.5)
- La signification d'un opérateur peut changer en fonction du type des opérandes (le cas de Java)
  - l'opérateur + avec des entiers effectue l'addition, 3+6 vaut 9
  - avec des chaînes de caractères, il effectue la **concaténation** "bonjour" + " tout le monde" vaut "bonjour tout le monde"

Expression : remarques

- Pour le langage C, si x et y sont entiers, x/y est une division entière alors que si l'un des deux ne l'est pas la division est réelle
- x+y/z : est une expression arithmétique dont le type dépend des types de x, y et z
- (x>y) | | !(x=y+1) : est une expression booléenne (**| |** dénote l'opérateur logique ou et ! dénote la négation)

Expression : remarques

- Avant d'utiliser une variable dans une expression, il est nécessaire qu'une valeur lui ait été affectée.
- La valeur de l'expression est évaluée au moment de l'affectation
  - **x ← 4**
  - **y ← 6**
  - **z ← x + y**
  - **Ecrire(z)** ➔ **10**
  - **y ← 20**
  - **Ecrire(z)** ➔ **10 la modification de y après affectation n'a aucun effet sur la valeur de z**

Les opérateurs booléens

- Associativité des opérateurs **et** et **ou**
  a et (b et c) = (a et b) et c
- Commutativité des opérateurs et et ou
  a et b = b et a
  a ou b = b ou a
- Distributivité des opérateurs et et ou
  a ou (b et c) = (a ou b) et (a ou c)
  a et (b ou c) = (a et b) ou (a et c)

Les opérateurs booléens

- Involution (homographie réciproque) : non non a = a
- Loi de Morgan : non (a ou b) = non a et non b
  non (a et b) = non a ou non b
- Exemple :
  soient a, b, c et d quatre entiers quelconques :
  (a<b)| |((a>=b)&&(c==d)) ⇔ (a<b)| |(c==d)
  car (a<b)| |(!(a<b)) est toujours vraie

Tables de vérité

C1	C2	C1 et C2	C1 ou C2	C1 XOR C2
Vrai	Vrai	Vrai	Vrai	Faux
Vrai	Faux	Faux	Vrai	Vrai
Faux	Vrai	Faux	Vrai	Vrai
Faux	Faux	Faux	Faux	Faux

C1	Non C1
Vrai	Faux
Faux	Vrai

Chap3: La lecture et l'écriture

Algorithmique

Les instruction de Lecture et d'Écriture

Les instructions d'entrées et sorties

- Soit le programme suivant qui calcule le carré d'un nombre, par exemple 7 :

```
Algorithme  Carre
Variables   A : entier;
Début
     A ← 12^2 ;
Fin
```

- Si on veut calculer le carré d'une autre valeur, il faut réécrire un autre programme !
- Solution : l'utilisateur doit entrer la valeur souhaitée.
- On parle d'opérations de lecture

Les instructions d'entrées et sorties : lecture et écriture

- 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 (la saisie).
  - En pseudo-code, on note: **lire (var);**
- Lorsque le programme rencontre une instruction de lecture (lire(var)), il passe la main à l'utilisateur pour la saisie de la valeur de var. Ce dernier doit appuyer sur la touche Entrée pour valider cette entrée.

Lecture et Ecriture des données

Remarque: lire (var); se déroule en trois étapes :

1) Le programme s'arrête lorsqu'il rencontre une instruction Lire et ne se poursuit qu'après la saisie de l'entrée attendue par le clavier.
2) La touche Entrée signale la fin de la saisie.
3) La machine place la valeur entrée au clavier (ou saisie) dans la zone mémoire nommée 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 taper (sinon longue attente).

Les instructions d'entrées et sorties : lecture et écriture

- **L'écriture** permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier)
  • En pseudo-code, on note: **écrire (liste d'expressions)**
  La machine affiche les valeurs des expressions décrite dans la liste d'expressions.
- Ces instructions peuvent être des variables ayant des valeurs, des nombres ou des commentaires sous forme de chaines de caractères.
- Exemple : écrire(a, b+2, "Message");

Exemple : lecture et écriture

Écrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui calcule et affiche le carré de ce nombre.

```
Algorithme Calcul_du_Carre
Rôle : calcul du carre
Données : un entier
Résultats : le carre du nombre
variables A, B : entier
Début
   écrire("entrer la valeur de A ");
   lire(A);
   B ← A*A;
   écrire("le carré de ", A, " est :", B);
Fin
```

Exercice : lecture et écriture

Écrire un algorithme qui permet d'effectuer la saisie d'un nom, d'un prénom et affiche ensuite le nom complet

```
Algorithme AffichageNomComplet
...
variables Nom, Prenom, Nom_Complet : chaîne de caractères
Début
    écrire("entrez le nom");
    lire(Nom);
    écrire("entrez le prénom");
    lire(Prenom);
    Nom_Complet ← Nom & " " & Prenom;
    écrire("Votre nom complet est : ", Nom_Complet);
Fin
```


Langage C

Présentation générale et instructions de base

Langage C

- Créé en 1972 (D. Ritchie et K. Thompson), est un langage rapide et trés populaire et largement utilisé.
- Le C++ est un langage orienté objet créé à partir du C en 1983.
- Le langage C a inspiré de nombreux langages :
  - C++, Java, PHP, ... leurs syntaxes sont proches de celle de C
- Le Langage C est un bon acquis pour apprendre d'autres langages

Premier programme en C

```
#include <stdio.h>

void main()
{
printf("Mon programme !\n");
}
```

bibliothèque

Point d'entré du programme

Instruction

Langage C : Généralités

- Chaque instruction en C doit se terminer par **;**
- Pour introduire un texte en tant que **commentaire**, il suffit de précéder la ligne par // (le texte est alors ignoré par le compilateur de C)
- Il est aussi possible d'écrire des **commentaires sur plusieurs lignes** en utilisant les symboles (/* ..*/)

```
/* exemple sur ligne 1
   exemple sur ligne 2 */
```


Langage C : nom et type des variables

- Le **nom d'une variable** peut être une combinaison de lettres et de chiffres, mais qui commence par une lettre, qui ne contient pas d'espaces et qui est différente des mots réservés du langage C
- Les principaux types définis en C sont :
  - **char** (caractères),
  - **int** (entier),
  - **short** (entiers courts),
  - **long** (entiers longs),
  - **float** (réel),
  - **double** (réel grande précision),
  - **long double** (réel avec plus de précision),
  - **unsigned int** (entier non signé)

Langage C : nom et type des variables

- Déclaration d'une variable
  - Type nom_de_la_variable [= valeur] ;
- Exemple:
  - int nb;
  - float pi = 3.14;//déclaration et initialisation
  - char c = 'x';
  - long a, b, c;
  - double r = 7.1974851592;

Langage C: l'affectation

- Le symbole d'**affectation** ← se note en C avec **=**

  exemple : i= 1; j= i+1;

- **Attention :** en C, le test de l'égalité est effectuée par l'opérateur ==

  a==b ; est une expression de type logique (**boolean**) qui est vrai si les deux valeurs a et b sont égales et fausse sinon

Langage C : affichage d'une variable

- **printf("format de l'affichage",** var**)** permet d'afficher la valeur de la variable var (c'est l'équivalent de **écrire** en pseudo code).
- **printf(**"chaine"**)** permet d'afficher la chaîne de caractères qui est entre guimets " "

  int a=1, b=2; printf("a vaut :%d et b vaut:%d \n ", a, b);

  a vaut 1 et b vaut 2
- **float r= 7.45; printf(" le rayon =%f \n ",r);**
- **Autres formats :**
  - **%c : caractère**
  - **%lf : double**
  - **%s : chaine de caractères**
  - **%e : réel en notation scientifique**

Langage C : affichage d'une variable

- **Affichage de la valeur d'une variable en C++**
  - cout <<chaîne 1 <<variable 1<<chaîne 2 <<variable 2;
  - Exemple
    - int i =2; int j = 20;
    - cout <<"i vaut:" << i <<"j vaut:"<<j <<'\n';
    - float r = 6.28;
    - cout<<"le rayon = "<< r <<'\n';

Langage C : lecture d'une variable

- Lecture d'une variable n de type entier:
- Syntaxe : scanf("%d ",&n); lit la valeur tapé par l'utilisateur au clavier et elle la stocke dans la variable n.
- Comme pour printf, le premier argument est une chaine de caractères qui donne le format de la lecture. Cette chaine ne peut contenir que des formats, pas de messages.
- Attention : notez la présence du caractère & devant n (adresse associée à la variable n) et ce n'est pas équivaut à scanf("%d", n);

Langage C : lecture d'une variable

- **lecture d'une variable en C++**
  - cin>>var;
  - Exemple
    - int i ;
    - cout <<"entrez i "<<'\n';
    - cin>>i;
    - float r ;
    - cout<<"entrez le rayon r " <<'\n';
    - cin>>r;

Chap4: Les instructions conditionnelles

Algorithmique 1

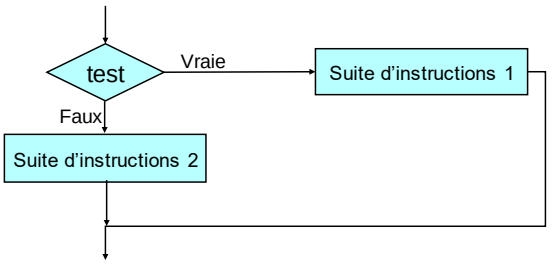
Les instructions conditionnelles

Tests: instructions conditionnelles

- Définition : une condition est une expression écrite entre parenthèse à valeur booléenne.
- Les instructions conditionnelles servent à n'exécuter une instruction ou une séquence d'instructions que si une condition est vérifiée.
- En pseudo-code :

Si condition alors
instruction ou suite d'instructions1
Sinon
instruction ou suite d'instructions2
Finsi

instructions conditionnelles



Instructions conditionnelles

- Remarques :
  - La condition ne peut être que vraie ou fausse
  - Si la condition est vraie alors seules les instructions1 sont exécutées.
  - Si la condition est fausse seules les instructions2 sont exécutées.
  - La condition peut être une expression booléenne simple ou une suite composée d'expressions booléennes.
- La partie Sinon est optionnelle, on peut avoir la forme simplifiée suivante:

Si condition **alors**
    instruction ou suite d'instructions1
Finsi

Si...Alors...Sinon : exemple

```
Algorithme  ValeurAbsolue1
Rôle : affiche la valeur absolue d'un entier
Données : la valeur à calculer
Résultat : la valeur absolue
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
```


Si...Alors : exemple

```
Algorithme ValeurAbsolue2
...
Variable x, y : réel
Début
    Ecrire (" Entrez un réel : " );
    Lire (x);
    y← x;
    Si (x < 0) alors
         y ← -x;
    Finsi
    Ecrire ("la valeur absolue de ", x, "est:",y);
Fin
```

Exercice (tests)

Écrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui teste et affiche s'il est divisible par 7 ou non

```
Algorithme Divsible_par7
...
Variable n : entier
Début
    Ecrire (" Entrez un entier : ");
    Lire (n)
    Si (n%7=0) alors
        Ecrire (n," est divisible par 7");
    Sinon
        Ecrire (n," n'est pas divisible par 7");
    Finsi
Fin
```


Langage C : syntaxe des tests

Écriture en pseudo code	Traduction en C
Si condition **alors** instructions **Finsi**	**if** (condition) **{** instructions; **}**
Si condition **alors** instructions1 **Sinon** instructions2 **Finsi**	**if** (condition) **{** instructions1; **} else {** instructions2; **}**

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.

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 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 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
```

Tests imbriqués

Remarque : dans l'exemple 2 on a fait trois tests systématiquement alors que dans l'exemple 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 disques compacts (CDs) dans espace de vente varie selon le nombre à acheter:
  5 DH l'unité si le nombre de CDs à acheter est inférieur à 10,
  4 DH l'unité si le nombre de CDS à acheter est compris entre 10 et 20 et 3 DH l'unité si le nombre de CDs à acheter est au-delà de 20.
- Écrivez un algorithme qui demande à l'utilisateur le nombre de CDs à acheter, qui calcule et affiche le prix à payer

Tests imbriqués : corrigé

```
Variables unites : entier
          prix : réel
Début
   Ecrire ("Nombre d'unités : ");
   Lire (unites);
   Si unites < 10 Alors
        prix ← unites*5;
   Sinon Si unites < 20  alors prix ← unites*4;
         Sinon  prix ← unites*3;
         Finsi
   Finsi
   Ecrire ("Le prix à payer est : ", prix);
Fin
```

Tests : remarques

- Un sinon se rapporte toujours au dernier si qui n'a pas encore de sinon associé
- Il est recommandé de structurer le bloc associé à si et celui associé à sinon
- Exemple :
  - **Lire(a);**
  - **x ←1;**
  - **Si (a>= 0) alors**
    si (a==0) alors x ←2 ;
    sinon x ←3;
    finsi
    finsi
    ecrire(x) ➔ **a : -1 0 1**
    affichage : 1 2 3

L'instruction cas

- Lorsque l'on doit comparer une **même** variable avec plusieurs valeurs, comme par exemple :

```
si a=1 alors instruction1
sinon si a=2 alors instruction2
        sinon si a=4 alors instruction4
                sinon . . .
                finsi
        finsi
finsi
```

- On peut remplacer cette suite de si par l'instruction cas

L'instruction cas

- Sa syntaxe en pseudo-code est :

```
cas où v vaut
      v1 : action1
      v2 : action2
      . . .
      vn : actionn
  autre : action autre
fincas
```

v1,. . . ,vn sont des **constantes** de type **scalaire** (entier, naturel, énuméré, ou caractère)

action i est exécutée si v = vi (on quitte ensuite l'instruction cas)

action autre est exécutée si quelque soit i, v ≠ vi

L'instruction cas : exemple 1

```
…
Variables c : caractère
Début
Ecrire("entrer un caractère");
Lire (c);
Si((c>='A') et (c<='Z')) alors
    cas où c vaut
          'A','E', 'I', 'O', 'U', 'Y' : ecrire(c, "est une voyelle majuscule");
                                autre : ecrire(c, " est une consonne majuscule ");
    fincas
 sinon ecrire(c, "n'est pas une lettre majuscule");
Finsi
Fin
```


L'instruction cas : exemple 2

```
…
Variables saison : entier;
Début
écrire(" Entrez une valeur entre 1 et 12 : ");
lire(saison);
Cas où saison vaut
          3..5 : écrire(" Printemps");
          6..8 : écrire("Eté");
     12, 1..2 : écrire("Hiver");
        autre : écrire("Automne");
finCas
Fin
```

Chap5: Les instructions itératives (les boucles)

Algorithmique 1

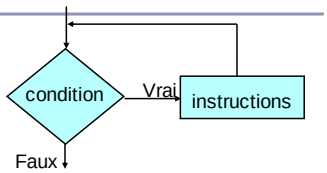
Les instructions Itératives : les boucles

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 non 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) faire
instructions
FinTantQue



- La condition (dite condition de contrôle de la boucle) est évaluée avant chaque itération.
- Si la condition est vraie, on exécute les instructions (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, …

Les boucles Tant que

- Si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après FinTantQue
- Il est possible que les instructions à répéter ne soient jamais exécutées.
- 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 la condition

Les boucles Tant que : remarques

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

Attention aux boucles infinies

- Exemple de boucle infinie :

```
i ← 1 ;
TantQue (i > 0) faire
    i ← i+1 ;
FinTantQue
```

correction

```
i ← 1;
TantQue ( i <100) faire
    i ← i+1;
FinTantQue
```


Boucle Tant que : exemple1

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

```
…
Variable  C : caractère
Debut
    Écrire (" Entrez une lettre majuscule ");
    Lire (C);
    TantQue ((C < 'A') ou (C > 'Z')) faire
        Ecrire ("Saisie erronée. Recommencez");
        Lire (C);
    FinTantQue
    Ecrire ("La saisie est valable ");
Fin
```

Tant que : exemple2

- En investissant chaque année 10000DH à intérêts composés de 7%, après combien d'années serons nous millionnaire ?

```
Variables capital :réel
          nbAnnees : entier
Debut capital ←0.0;     nbAnnes ←0;
TantQue (Capital < 1000000) faire
     capital ← capital+10000;
     nbAnnees++;
     capital ←(1+0.07)*capital;
FinTantQue
Fin
```


Boucle Tant que : exemple3

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

forme 1:

```
   Variables som, i : entier
Debut
        i ← 0;
        som← 0;
   TantQue (som <=100) faire
        i ← i+1;
        som ← som+i;
   FinTantQue
   Ecrire (" La valeur cherchée est N= ", i);
Fin
```


Boucle Tant que : exemple3

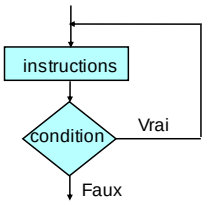
Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

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

```
Variables som, i : entier
Debut
      som ← 0;
      i ← 1;
      TantQue (som <=100) faire
            som ← som + i;
            i ← i+1;
      FinTantQue
      Écrire (" La valeur cherchée est N= ", i-1);
Fin
```

Les boucles Répéter … jusqu'à …

Répéter
instructions
tantQue condition



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

Boucle Répéter jusqu'à : exemple 1

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

```
Variables som, i : entier
Debut
    som ← 0;
    i ← 0;
    Répéter
        i ← i+1;
        som ← som+I;
    tantQue( som <= 100)
    Ecrire (" La valeur cherchée est N= ", i);
Fin
```


Boucle Répéter jusqu'à : exemple 2

Ecrire un algorithme qui compte le nombre de bits nécessaires pour coder en binaire un entier n.

Solution :

```
Variables i, n, nb : entiers
Debut
    Ecrire(" Entrer la valeur de n :");
    lire(n);
    i ← n;
    nb ← 0 ;
    Répéter
        i ← i/2;
        nb ← nb + 1;
    tantQue(i>0)
    Ecrire("Pour coder ",n," en binaire il faut ",nb, "bits");
Fin
```

Les boucles Tant que et Répéter jusqu'à

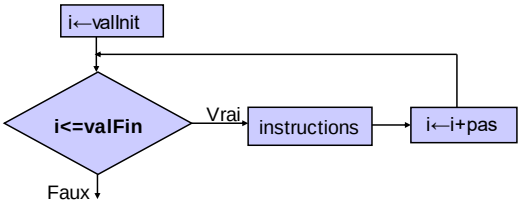
- Différences entre les boucles TantQue et Répéter tantQue :
  - La séquence d'instructions est exécutée au moins une fois dans la boucle Répéter tantQue, alors qu'elle peut ne pas être exécutée dans le cas du Tant que.

  - Dans les deux cas, la séquence d'instructions est exécutée si la condition est vraie.

  - Dans les deux cas, la séquence d'instructions doit nécessairement faire évoluer la condition, faute de quoi on obtient une boucle infinie.

Les boucles Pour

Pour <var> **← valInit à valFin** [par <**pas**>] **faire**
  instructions
FinPour



Les boucles Pour

- Remarque : le nombre d'itérations dans une boucle Pour est connu avant le début de la boucle.
- **var** est une variable compteur 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 valFin - vallnit+1.
- **vallnit** et **valFin** peuvent être des valeurs, des variables définies avant le début de la boucle ou des expressions de même type que var.

Déroulement des boucles Pour

1) La valeur valInit est affectée à la variable var (compteur)

2) On compare la valeur de var et la valeur de valFin :

a) Si la valeur de var est > à valFin dans le cas d'un pas positif (ou si var est < à valFin pour un pas négatif), on sort de la boucle et on continue avec l'instruction qui suit FinPour

Déroulement des boucles Pour

b) Si var est <= à valFin dans le cas d'un pas positif (ou si var est >= à valFin pour un pas négatif), instructions seront exécutées

i. Ensuite, la valeur du var 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 var et valFin est de nouveau effectuée, et ainsi de suite …

Boucle Pour : exemple 1 (forme 1)

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 respectivement les valeurs de x et n ");
    Lire (x, n);
    puiss ← 1;
    Pour i ←1 à n faire
            puiss← puiss*x;
    FinPour
    Ecrire (x, " à la puissance ", n, " est égal à ", puiss);
Fin
```

Boucle Pour : exemple1 (forme 2)

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul (forme 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 ← 1;
    Pour i← n à 1 par pas -1 faire
        puiss← puiss*x;
    FinPour
    Ecrire (x, " à la puissance ", n, " est égal à ", puiss);
Fin
```


Boucle Pour : remarques

- Il faut éviter de modifier la valeur du compteur var (et de valFin) 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 ← 1 à 5 faire
        i ←i -1;
        écrire(" i = ", i);
    Finpour
```


Lien entre Pour et TantQue

- La boucle Pour est un cas particulier de TantQue (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 ← valInit à valFin [par pas] faire
    instructions
FinPour
```

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

```
compteur ←valInit
TantQue compteur <= valFin faire
        instructions
        compteur ← compteur+pas
FinTantQue
```

Lien entre Pour et TantQue: exemple 1

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul (forme avec TantQue)

```
Variables x, puiss : réel
          n, i : entier
Debut
    Ecrire (" Entrez respectivement les valeurs de x et n ");
    Lire (x, n);
    puiss ← 1; i ← 1;
    TantQue (i<=n) faire
        puiss← puiss*x;
        i ← i+1;
    FinTantQue
    Ecrire (x, " à la puissance ", n, " est égal à ", puiss);
Fin
```


Boucles : exercice

- Ecrire un algorithme qui compte le nombre de 1 dans la représentation binaire de l'entier n.

Solution :

```
Variables i, n, poids : entiers
Debut
    Ecrire(" Entrer la valeur de n :");
    lire(n);
    i ← n;
    poids ← 0 ;
    TantQue(i<>0) faire
        si (i mod 2 == 1) alors poids ← poids + 1;
        i ← i/2;
    FinTantQue
    Ecrire("Pour l'entier",n," le poids est : ",poids);
Fin
```


Sortie de la boucle TantQue

- Interpréter(et bien comprendre!) l'arrêt des itérations à la sortie d'une boucle.
- **TantQue** <cond> **faire …**
  - À la sortie : **non**(<cond>) *est vrai*
  - donc si cond = p **et** q à la sortie : **non**(p **et** q) c'est à dire **non** p **ou non** q
- **Exemple**: avec **<cond>** égal à: val !=STOP **et** nbVal<MAX **non(<cond>)** égal à : val ==STOP **ou** nbVal≥MAX

Boucles imbriquées

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

- **Exemple:**

```
Pour i ← 1 à 5 faire
    Pour j ← 1 à i faire
        écrire("O");
    FinPour
    écrire("K");
FinPour
```

Exécution

OK
OOK
OOOK
OOOOK
OOOOOK

Boucles successives

- Il y a une grande différence entre les boucles imbriquées d'ordre (n x m) et les boucles successives d'ordre (n+m).

```
Variables Tic, Tac :entiers;
Début
Pour Tic ←1 à 5 faire
        Ecrire ("Il a été là.");
FinPour
Pour Tac ← 1 à 6 faire
        Ecrire ("Il sera ici.");
FinPour
Fin
```


Choix d'un type de boucle

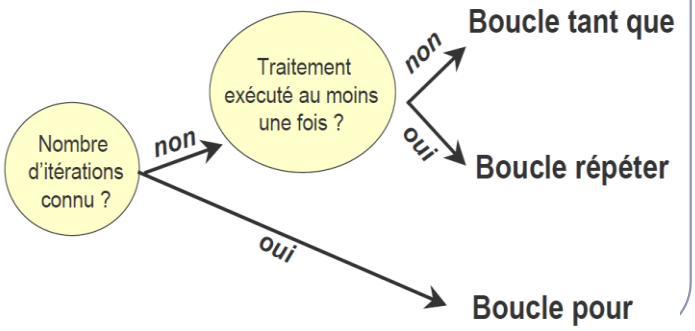
- Si on peut déterminer le nombre d'itérations avant l'exécution de la boucle, il est plus naturel d'utiliser *la boucle Pour.*

- 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 tantQue.*

Choix d'un type de boucle

- Pour le choix entre *TantQue* et *répéter tantQue :*
  - 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 tantQue.*

Choix du type de boucle ?



Boucles « TantQue » et « Répéter TantQue »

Remarques: Soient T une condition et R l'action. Alors il y a équivalence entre les boucles Tant Que et Répéter TantQue .

La syntaxe est la suivante:

TantQue T faire R FinTantQue	~	Si T alors Répéter R TantQue T FinSi
Et		
Répéter R TantQue T	~	R TantQue T faire R FinTantQue

Boucle «Pour » : exercice

1- Écrire un algorithme permettant de déterminer le $N^{ème}$ terme d'une suite numérique connaissant son premier terme et ses coefficients a et b et tels que:

$$U_n = a * U_{n-1} + b \qquad \forall \quad 1 \leq n \leq N$$

2- Écrire un algorithme permettant de définir le rang N et le terme correspondant de la suite tels que $U_N > 1000$;

Solution :

1) Le nombre d'itérations est connu : Boucle Pour

```
Variable    N, i : Entier
Variable    a, b, S : Réel
DEBUT
    Écrire (" Saisir la valeur de N: ");
    Lire (N) ;
    Écrire ("Saisir le premier terme et les coefs a et b:") ;
    Lire (S , a, b );
    Pour   i ← 1  à  N  faire
           S  ←———  a * S  +  b
     Fin Pour
      Écrire (" La somme de la série est : ", S )
FIN
```


Solution :

2) Le nombre d'itérations inconnu : Boucle Répéter Tant Que

```
Variable    N : Entier
Variable    a, b, S : Réel
DEBUT
    Écrire ("Saisir le premier terme et les coefs a et b:");
    Lire (S , a, b) ;
    N ←—— 0;
      Répéter
          S  ←———  a * S  +  b;
          N ←—— N + 1;
      TantQue  S <= 1000
      Écrire (" La somme de la série est : ", S);
     Écrire ("Le rang est : ", N );
FIN
```

Langage C : syntaxe des boucles

Écriture en pseudo code	Traduction en C
TantQue condition Instructions **FinTantQue**	**while(** condition) { instructions; **}**
Pour i ← *v1* **à** *v2* par **pas** *p* instructions **FinPour**	**for(** i=v1;i<=v2;i=i+p){ instructions; **}**
Répéter **instructions** **TanTQue condition**	**do**{ **instructions;** **} while(condition)**

Chap6: Les tableaux

Les tableaux

Tableaux : introduction

- Supposons que l'on veut calculer le nombre d'étudiants ayant une note supérieure à 10 pour une classe de 20 étudiants.
- Jusqu'à présent, le seul moyen pour le faire, c'est de déclarer 20 variables désignant les notes **N1, …, N20**:
  - La saisie de ces notes nécessite 20 instructions lire(Ni).
  - Le calcul du nombre des notes>10 se fait par une suite de tests de 20 instructions Si :

```
nbre ← 0;
Si (N1 >10) alors nbre ←nbre+1 FinSi
…
Si (N20>10) alors nbre ←nbre+1 FinSi
```

cette façon n'est pas très pratique

Tableaux

- C'est pourquoi, les langages de programmation offrent la possibilité de rassembler toutes ces variables dans **une seule structure de donnée** appelée **tableau qui est facile à manipuler.**
- 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.

Tableaux

- 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[20] : réel
- On peut définir des tableaux de tous types : tableaux d'entiers, de réels, de caractères, de booléens, de chaînes de caractères, …

Les tableaux

- Les tableaux à une dimension ou vecteurs :

  variable **tableau tab[10] : entier**

0	1	2	3	4	5	6	7	8	9
45	54	1	-56	22	134	49	12	90	-26

- Ce tableau est de longueur 10, car il contient 10 emplacements.
- Chacun des dix nombres du tableau est repéré par son rang, appelé indice.
- Pour accéder à un élément du tableau, il suffit de préciser entre crochets l'indice de la case contenant cet élément.

  Pour accéder au 5ème élément (22), on écrit : tab[4]

Les tableaux

- Pour accéder au **i^{ème}** élément, on écrit tab[i-1] (avec 0<=i<10)
- 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 utiliser en pseudo-code).
- Dans ce cas, tab[i] désigne l'élément i+1 du tableau tab.

Tableaux : remarques

- 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.
- Les éléments d'un tableau s'utilisent comme des variables.

Tableaux : accès et modification

- Les instructions de lecture, écriture et affectation s'appliquent aux tableaux comme aux variables.
- Exemples :

  x ← tab[0];

  La variable x prend la valeur du premier élément du tableau (45 selon le tableau précédent).

  tab[6] ← 43;

  Cette instruction a modifié le contenu du 7ème élément du tableau (43 au lieu de 49).

Relation entre tableaux et boucles

- Les boucles sont extrêmement utiles pour les algorithmes associés aux tableaux.
- Pour parcourir les éléments du tableau selon l'ordre croissant (ou décroissant) des indices, on utilise des boucles.
- Le traitement de chacun des éléments étant souvent le même, seule la valeur de l'indice est amenée à changer.
- Une boucle est donc parfaitement adaptée à ce genre de traitements.

Tableaux : exemple 1

- Pour le calcul du nombre d'étudiants ayant une note supérieure à 12 avec les tableaux, on peut écrire :

```
…
Constante          N=20 : entier
Variables                i, nbre : entier
        tableau notes[N] : réel
Début
    nbre ← 0;
    Pour i ← 0 à N-1 faire
        Si (notes[i] >12) alors
            nbre ←nbre+1;
        FinSi
    FinPour
    écrire ("le nombre de notes supérieures à 12 est : ", nbre);
Fin
```


Tableaux : exemple 2

- Le programme suivant comporte la déclaration d'un tableau de 20 réels (les notes d'une classe), on commence par effectuer la saisie des notes, et en suite on calcul la moyenne des 20 notes et on affiche la moyenne :

```
…
Constante   Max =20 : entier
variables tableau    Notes[Max], i, somme, n : entier
                          moyenne          : réel
début
    ecrire("entrer le nombre de notes :"); lire(n);
        /* saisir les notes */
    pour i ← 0 à n-1 faire
        ecrire("entrer une note :");
        lire(Notes[i]);
    finpour
        /* calculer la moyenne des notes */
    somme ← 0 ;
    pour i ← 0 à n-1 faire
        somme ← somme + Notes[i] ;
    finPour
    moyenne = somme / n;
        /* affichage de la moyenne */
    ecrire("la moyenne des notes est :",moyenne)
fin
```


Tableaux : saisie et affichage

- Saisie et affichage des éléments d'un tableau :

```
Constante  Max=200 : entier
variables      i, n       : entier
      tableau Notes[max] : réel
ecrire("entrer la taille du tableau :") ; lire(n);
/* saisie  */
      Pour i ← 0 à n-1 faire
            écrire ("Saisie de l'élément ", i + 1);
            lire (T[i] );
      FinPour
/* affichage  */
      Pour i ← 0 à n-1 faire
            écrire ("T[",i, "] =", T[i]);
      FinPour
```

Les tableaux : Initialisation

Le bloc d'instructions suivant initialise un à un tous les éléments d'un tableau de n éléments :

- InitTableau

```
début
   pour i ← 0 à n-1 faire
      tab[i] ← 0;
   fpour
fin
```


Tableaux : Exercice

Que produit l'algorithme suivant ?

```
Variable Tableau F[10], i : entier
  début
      F[0]← 1;
      F[1]← 1;
      écrire(F[0],F[1]);
      pour i ← 2 à 9 faire
          F[i]← F[i-1]+F[i-2];
          écrire(F[i]);
      finpour
  fin
```


Tableaux : syntaxe en C

- En langage C, un tableau se déclare comme suit :
  type nom_tableau[dimension];
  dimension : doit être une constante
  - Exemple : **int t[100] ;**
- La taille n'est pas obligatoire si le tableau est initialisé à sa création.
  - Exemple : **int dixPuissance[ ] = { 0, 1, 10, 100, 1000, 10000 } ;**
- Ne pas confondre :
  - taille maximale : **dimension** (une constante)
  - taille effective : nombre de cases réellement utilisées lors de la manipulation du tableau (une variable)

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**[dim1] [dim2] : type;**

Tableaux à deux dimensions

- **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+1 et de la colonne j+1
- Les tableaux peuvent avoir n dimensions.

Les tableaux à deux dimensions

- La matrice A dans la déclaration suivante :

  variable **tableau** A**[3][7] : réel**

  peut être explicitée comme suit : les éléments sont rangées dans un tableau à deux entrées.

	0	1	2	3	4	5	6
0	12	28	44	2	76	77	32
1	23	36	51	11	38	54	25
2	43	21	55	67	83	41	69

Ce tableau a 3 lignes et 7 colonnes. Les éléments du tableau sont repérés par leur numéro de ligne et leur numéro de colonne désignés en bleu. Par exemple A[1][4] vaut 38.

Exemples : lecture d'une matrice

- La saisie des éléments d'une matrice :
- **Constante** N=100 :entier
  Variable i, j, m, n : entier
  tableau A[N][N] : réel
  Début
  ecrire("entrer le nombre de lignes et le nombre de colonnes :") ;
  lire(m, n);
  Pour i ← 0 à m-1 faire
  écrire ("saisie de la ligne ", i + 1);
  Pour j ← 0 à n-1 faire
  écrire ("Entrez l'élément : ligne ", i+1, " et colonne ", j+1);
  lire (A[i][j]);
  FinPour
  FinPour
  Fin

Exemples : affichage d'une matrice

- Affichages des éléments d'une matrice :
- **Constante** N=100 : entier
  Variable i, j, m, n : entier
  tableau A[N][N], B[N][N], C[N][N] : réel

  Début
  ecrire("entrer le nombre de lignes et le nombre de colonnes :") ;
  lire(m, n);
  Pour i ← 0 à m-1 faire
  Pour j ← 0 à n-1 faire
  écrire ("A[",i, "] [",j,"]=", A[i][j]);
  FinPour
  FinPour
  Fin

Initialisation de matrice

- Pour initialiser une matrice on peut utiliser par exemple les instructions suivantes :

T_1[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
T_2[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
T_3[4][4] = {{1,2,3}, {4,5,6}, {7,8,9}};
T_4[4][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Exemples : somme de deux matrices

- Procédure qui calcule la somme de deux matrices :

```
Constante  N =100 :entier
Variable                i, j, n : entier
                tableau A[N][N], B[N][N], C[N][N] : réel
Début
 ecrire("entrer la taille des matrices :") ;
 lire(n);
 Pour i ← 0 à n-1 faire
      Pour j ← 0 à n-1 faire
                C[i][j] ← A[i][j]+B[i][j];
      FinPour
 FinPour
Fin
```


Exemples : produit de deux matrices

```
constante N=20 : entier
variables Tableau A[N][N],B[N][N],C[N][N],  i, j, k, n, S : entier
début
        écrire("donner la taille des matrices(<20) :");
        lire(n);
                                /* lecture de la matrice A */
        pour i ← 0 à n-1 faire
                écrire("donner les éléments de la ",i+1," ligne:");
                pour j ← 0 à n-1 faire
                        lire(A[i][j]) ;
                finpour
        finpour
                                /* lecture de la matrice B */
        pour i ← 0 à n-1 faire
                écrire("donner les éléments de la ",i+1," ligne:");
                pour j ← 0 à n-1 faire
                        lire(B[i][j]) ;
                finpour
        finpour
```


Exemples : produit de deux matrices (suite)

```
                /* le produit de C = A * B */
        pour i ← 0 à n-1 faire
                pour j ← 0 à n-1 faire
                        S ← 0;
                        pour k ← 0 à n-1 faire
                                S ← S + A[i][k]*B[k][j] ;
                        finpour
                        C[i][j] ← S;
                finpour
        fipour
                /* affichage de la matrice de C */
        pour i ← 0 à n-1 faire
                pour j ← 0 à n-1 faire
                        écrire(C[i][j],"  ") ;
                finpour
                écrire("\n");       /* retour à la ligne */
        finpour
fin
```

Chap7: Les fonctions et les procédures

Les fonctions et les procédures

Notion de réutilisabilité

- Pour l'instant, un programme est une séquence d'instructions mais sans **partage** des parties importantes ou utilisées plusieurs fois.
- Le bloc suivant peut être exécuté à plusieurs endroits

```
Répéter
    Ecrire("Entrez un nombre entre 1 et 100 : "); Lire(i);
TantQue ((i < 1) ou (i > 100));
```

- Bonne pratique : **Ne jamais dupliquer** un bloc de code **!**
- Ce qu'on veut recopier doit être mis dans une **fonction.**

Les procédures et les fonctions

- Résoudre le problème suivant :

  Écrire un programme qui affiche, en ordre croissant, les notes d'une classe suivies de la note la plus faible, de la note la plus élevée et de la moyenne.

  revient à résoudre les sous problèmes suivants :
  - Remplir un tableau des notes saisies par l'utilisateur
  - Afficher un tableau des notes
  - Trier un tableau de notes en ordre croissant
  - Trouver le plus petit réel d'un tableau
  - Trouver le plus grand réel d'un tableau
  - Calculer la moyenne d'un **tableau** de réels

Les procédures et les fonctions

Chacun de ces sous-problèmes devient un nouveau problème à résoudre.

Si on considère que l'on sait résoudre ces sous-problèmes, alors on sait "quasiment" résoudre le problème initial.

Donc écrire un programme qui résout un problème revient toujours à écrire des sous-programmes qui résolvent des sous parties du problème initial.

En algorithmique, il existe deux types de sous-programmes :

- Les fonctions
- Les procédures

Les procédures et les fonctions

```
Algorithme  TraitTableau;
variables   tableau tab[100] : réel;
 pPetit, pGrand, moyen : réel;
Début
   Saisire(tab);
   Afficher(tab);
   pPetit←plusPetitElements(tab);
   pGrand←plusGrandElements(tab);
   moyen ←calculerMoyen(tab);
   Trier(tab);
Fin
```


Les fonctions et les procédures

- Un programme long est souvent difficile à écrire et à comprendre.
- C'est pourquoi, il est préférable de le décomposer en des parties appelées **sous-programmes** ou **modules.**
- Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom.

Les fonctions et les procédures

Elles ont plusieurs avantages :

- Permettent d'éviter de réécrire un même traitement plusieurs fois. En effet, on fait appel à la procédure ou à la fonction aux endroits spécifiés.
- Permettent d'**organiser le code** et améliorent la **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.

Les fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat au programme appelant.**
- Le corps de la fonction est la portion de programme à réutiliser ou à mettre en évidence.
- Les paramètres de la fonction : (les «entrées», ou les «arguments») ensemble de variables extérieures à la fonction dont le corps dépend pour fonctionner.

Les fonctions

- Une fonction s'écrit en dehors du programme principal sous la forme:

```
Fonction nom_fonction (paramètres et leurs types) :
   type_fonction
Variables  // variables locales
Début
   Instructions;  //le corps de la fonction
   retourne;    //la valeur à retourner
FinFonction
```

- Le nom_fonction : désignant le nom de la fonction.
- type_fonction est le type du résultat retourné.
- L'instruction retourne sert à retourner la valeur du résultat.

Caractéristiques des fonctions

- La fonction est désignée par son nom.
- Une fonction ne modifie pas les valeurs de ses arguments en entrée.
- Elle se termine par une instruction de retour qui rend un résultat et un seul.
- Une fonction est toujours utilisée dans une expression (affectation, affichage,…) ayant un type compatible avec le type de retour de la fonction.
- On doit fournir une valeur pour chacun des arguments définis pour la fonction (certains langages acceptent des valeurs par défaut).

Les fonctions : exemples

Quel est le type du résultat ?

Comment appeler la méthode ?

Quels sont les paramètres

- La fonction max suivante retourne le plus grand des deux réels x et y fournis en arguments :

```
Fonction    max (x : réel, y: réel ) : réel
    variable  z : réel
    Début   z ←y;
            si (x>z) alors z ←x finsi;
            retourne (z);
    FinFonction
```

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

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

Que fait la méthode ?

Quel est le résultat ?

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, ...
- **Exemple: Algorithme exepmleAppelFonction**

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

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

Évaluation d'un appel de fonction

- L'évaluation de l'**appel :** f(arg1, arg2, ..., argN); d'une fonction définie par :

  typeR f(type1 x1, type2 x2, ..., typeN xN) { ... }

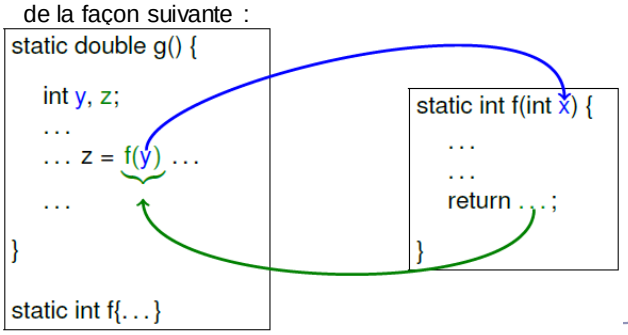
  s'effectue de la façon suivante :
- 1. Les expressions arg1, arg2, ..., argN passées en argument sont évaluées.
- 2. les valeurs correspondantes sont **affectées** aux paramètres x1, x2, ..., xN de la fonction f (variables locales à f).
- Concrètement, ces deux premières étapes reviennent à faire : x1←arg1; x2← arg2; ...;xN← argN;

Évaluation d'un appel d'une fonction

- 3. les instructions correspondantes au corps de la fonction f sont exécutées.
- 4. l'expression suivant la première commande return rencontrée est évaluée...
- 5. ...et retournée comme résultat de l'appel : cette valeur remplace l'expression de l'appel, c-à-d l'expression f(arg1, arg2, ..., argN);

L'appel d'une fonction

- L'évaluation de l'appel d'une méthode peut être schématisé de la façon suivante :



Les procédures

- Dans le cas où une tâche se répète dans plusieurs endroits du programme et elle ne calcule pas de résultats ou qu'elle calcule plusieurs résultats à la fois alors on utilise une **procédure** au lieu d'une fonction.
- Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien.**

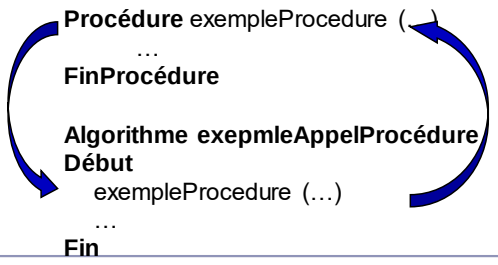
Les procédures

- Une procédure s'écrit en dehors du programme principal sous la forme :

  Procédure nom_procédure (paramètres et leurs types)
  Variables //locales
  Début
  Instructions constituant le corps de la procédure
  FinProcédure
- Remarque : une procédure peut ne pas avoir de paramètres

Appel d'une procédure

- Pour appeler une procédure dans un programme principale ou dans une autre procédure, il suffit d'écrire une instruction indiquant le nom de la procédure :



Appel d'une procédure

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**. Ils sont des variables locales à la procédure.
- Les paramètres placés dans l'appel d'une procédure sont appelés **paramètres effectifs**. ils contiennent les valeurs pour effectuer le traitement.

Paramètres d'une procédure

- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels.
- L'ordre et le type des paramètres doivent correspondre.
- Il existe deux modes de transmission de paramètres dans les langages de programmation :

La transmission par valeur et la transmission par adresse.

Transmission des paramètres

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

Transmission des paramètres : exemples

```
Procédure incrementer1 (x : entier par valeur, y : entier par adresse)
        x ← x+1;
        y ← y+1;
FinProcédure

Algorithme Test_incrementer1
variables  n, m : entier
Début
    n ← 3;
    m ← 3;
    incrementer1(n,  m) ;
    écrire (" n= ", n, " et m= ", m) ;
Fin
```

résultat :
n=3 et m=4

Remarque : l'instruction x ← 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 échange le contenu de deux variables :

```
Procédure Echange (x : réel par adresse, y : réel par adresse)
Variables  z : réel
        z ← x;
        x ← y;
        y ← z;
FinProcédure
```

Variables locales et globales

- 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 principale. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme.

Variables locales et globales

- 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.

Fonctions et procédures en langage C

- En C, une fonction prends N arguments et retourne une valeur de type.

Syntaxe : type arg_ret nom_f(type arg1, type arg2, …type argn)

{ instructions;
return expression; }

  - arg_ret est l'argument renvoyé par la fonction (instruction return)
  - nom_f est le nom de la fonction
  - arg1 …argn sont les arguments envoyés à la fonction.
- Une procédure est une fonction renvoyant void, dans ce cas return est appelé sans paramètre.

Fonctions et procédures en C

- L'ordre, le type et le nombre des arguments doivent être respectés lors de l'appel de la fonction.
- L'appel d'une fonction doit être située après sa déclaration ou celle de son prototype.
- Si la fonction ne renvoie rien alors préciser le type *void* (cette fonction est considérée comme une procédure)

Fonctions en C : exemple

```
int min(int a, int b);
void main()
{ int c;
/* entrez les valeurs de a et b */
   c= min(a, b) ;
    printf("le min de %d et %d est : %d \n", a, b, c);
}
int min(int a, int b)
{
     if (a <b) return a;
     else return b;
}
```


La récursivité

- Un module (fonction ou procédure) peut s'appeler lui-même: on dit que c'est un module **récursif**
- Un module peut comporter un ou plusieurs appels récursifs.
- La formulation d'une solution récursive décrit plusieurs éléments.
  - Les cas de base : ces cas sont les conditions d'arrêt de la chaîne des appels récursifs.
  - Les appels récursifs eux-mêmes.

La récursivité

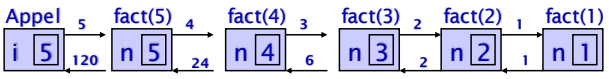
- La façon de formuler ces appels a un impact sur la convergence de la solution récursive.
  - Il faut que tout appel initial nous amène éventuellement à un des cas de base.
  - Pour ce faire, il faut que chaque appel nous rapproche des cas de base sans pour autant les dépasser.

Récursivité : exemple

- **Exemple** : Calcul du factorielle

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

- On résume la démarche utilisée par :
  n! = n × (n-1)!; n=n × (n-1) × (n-2)! ...
  - Le cas simple correspond à la valeur 0.
- L'image suivante illustre le calcul de 5! :



la récursivité généralise la structure de boucle

- Un module récursif comporte un effet de répétition. D'autre part, une boucle à compteur donne lieu à une situation de récursivité.
- Dans un module récursif, chaque appel récursif est l'analogue d'une itération de boucle.
- La différence entre une boucle et un module récursif provient du fait que l'on peut effectuer des traitements avant ou après l'appel récursif :
  - - Les traitements avant l'appel récursif représentent les instructions de la boucle.
  - - Les traitements après l'appel récursif sont laissés en suspend et sont placés sur une pile. C'est ce qui explique que l'on peut effectuer les choses à l'envers.

Fonctions récursives : exercice

- Écrivez 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
 début
        Si (n=1 OU n=0) alors  res ←1;
        Sinon
                res ← Fib(n-1)+Fib(n-2);
        Finsi
        retourne (res);
FinFonction
```


Fonctions récursives : exercice

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

```
Fonction Fib (n : entier ) : entier
Variables i, AvantDernier,  Dernier,  Nouveau  : entier
 Début
        Si (n=1 OU n=0) alors retourne (1);
        Finsi
        AvantDernier  ←1; Dernier ←1;
        Pour i allant de 2 à n
                Nouveau←  Dernier+  AvantDernier;
                AvantDernier  ←Dernier;
                Dernier ←Nouveau;
        FinPour
        retourne (Nouveau);
FinFonction
```

Remarque: 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 )
Début
        Si (n<>0) alors
                binaire (n/2);
                écrire (n mod 2);
        Finsi
FinProcédure
```

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 pouvoir calculer la factorielle par exemple.
- Cet effet de rebours est caractéristique de la programmation récursive.

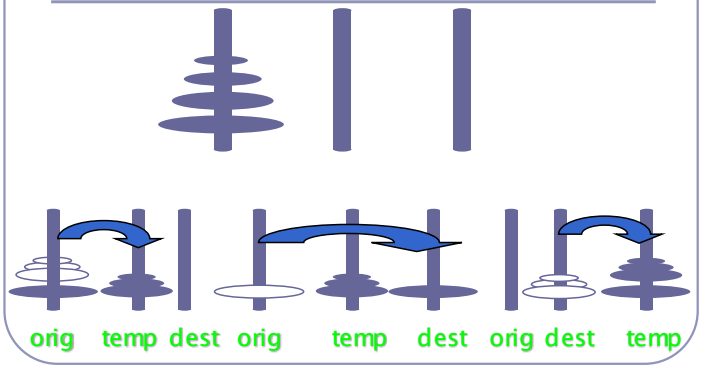
Les fonctions récursives : remarques

- 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 coûteuse en ressources machine. Car il faut créer autant de variable 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 indispensable.

Le problème des tours de Hanoï

- Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges. Sur ces tiges sont enfilés des disques de diamètres tous différents. Les seules règles du jeu sont que l'on ne peut déplacer qu'un seul disque à la fois, et qu'il est interdit de poser un disque sur un disque plus petit.
- Au début tous les disques sont sur la tige de gauche, et à la fin sur celle de droite.

Le problème des tours de Hanoï



Résolution du problème des tours de Hanoï

- L'algorithme suivant permet de résoudre le problème des tours de Hanoï.

Variables :

- **n** : numéro du disque à déplacer. Plus le numéro est grand, plus le disque est grand.
- **orig** : nom de la tige d'origine.
- **dest** : nom de la tige destination.
- **temp** : nom de la tige temporaire.

Résolution du problème des tours de Hanoï

```
Procédure hanoi (n : entier, orig, dest, temp : chaine de
   caracteres )
Début
   Si (n ≥ 1) alors
      hanoi (n-1, orig, temp, dest);
      écrire ("déplacer ", n, " de ", orig, " vers ", dest);
      hanoi (n-1, temp, dest, orig);
   Fin-si
FinProcédure
```

Résolution du problème des tours de Hanoï

Exécution avec trois disques :

- 1. Déplace un disque de la tige orig vers la tige dest
- 2. Déplace un disque de la tige orig vers la tige temp
- 3. Déplace un disque de la tige dest vers la tige temp
- 4. Déplace un disque de la tige orig vers la tige dest
- 5. Déplace un disque de la tige temp vers la tige orig
- 6. Déplace un disque de la tige temp vers la tige dest
- 7. Déplace un disque de la tige orig vers la tige dest

La programmation modulaire

Intérêt de la programmation modulaire :

- **Permettre une analyse descendante d'un problème** :
  - identifier les différents traitements contribuant au travail demandé.
  - organiser l'enchaînement des étapes.
- **Permettre une mise au point progressive, module par module.**

La programmation modulaire

- **Faciliter la maintenance des programmes**
  - modifier le traitement lui-même sans changer le rôle particulier d'un module.
- **Enrichir le langage algorithmique en ajoutant de nouveaux "mots" du langage**
  - notion de "boite à outils", de bibliothèques de composants logiciels réutilisables.

Méthodologie d'analyse d'un problème

- **Problème : écrire l'algorithme du jeu de Saute Mouton**

  Sur une ligne de NB cases, on place, à la suite et en partant de la gauche, des pions noirs puis des pions rouges séparés par une case vide unique. On pose autant de pions noirs que de pions rouges. La configuration de pions n'occupe pas nécessairement toute la ligne.

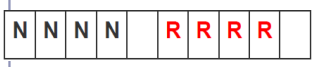
Méthodologie d'analyse d'un problème

- **But du jeu :**

  Déplacer tous les pions rouges vers la gauche (respectivement tous les pions noirs vers la droite), la case vide occupant à la fin du jeu la case du milieu de la configuration comme au départ.

- **Exemple :**

Configuration initiale



Configuration finale gagnante

Méthodologie d'analyse d'un problème

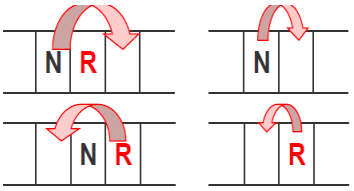
Règles du jeu:

- les pions **noirs** ne peuvent se déplacer que vers la droite.
- les pions **rouges** ne peuvent se déplacer que vers la gauche.
- un pion **noir** peut être déplacé à droite dans la case vide :
  - si la case vide est juste à droite de ce pion
  - s'il lui suffit de sauter par dessus un seul pion rouge, c'est-à-dire si entre la case vide et lui il n'y a qu'un seul pion rouge.
- un pion **rouge** peut être déplacé à gauche dans la case vide :
  - si la case vide est juste à gauche de ce pion.
  - s'il lui suffit de sauter par dessus un seul pion noir, c'est-à-dire si entre la case vide et lui il n'y a qu'un seul pion noir.

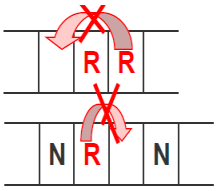
Méthodologie d'analyse d'un problème

- **Exemple :**

Coups permis



coups interdits



Méthodologie d'analyse d'un problème

Fonctionnement :

- On joue en donnant simplement la position du pion qu'on joue.
- La machine déplace le pion choisi si c'est possible.
- Le jeu s'arrête si on a gagné ou si on a atteint une situation de blocage.
- Dans ce cas on a perdu !

Méthodologie d'analyse d'un problème

- **Comment lire un énoncé de problème dans le but d'écrire l'algorithme correspondant :**
  - repérer les **données** proposées
  - repérer les **résultats** attendus
  - identifier les **contraintes** de la tâche
  - définir les **traitements** à réaliser

Choix des structures

- *Dans un premier temps, se demander quelles structure de données utiliser pour la représentation interne des données :*
- Le plateau de jeu ➔ **Plateau:** tableau de caractères
- **constante** NbMAX←10: entier /* *nombre max de pions d'une couleur*/*
- **Variable tableau** jeu [2 x NbMAX+1] : caractères

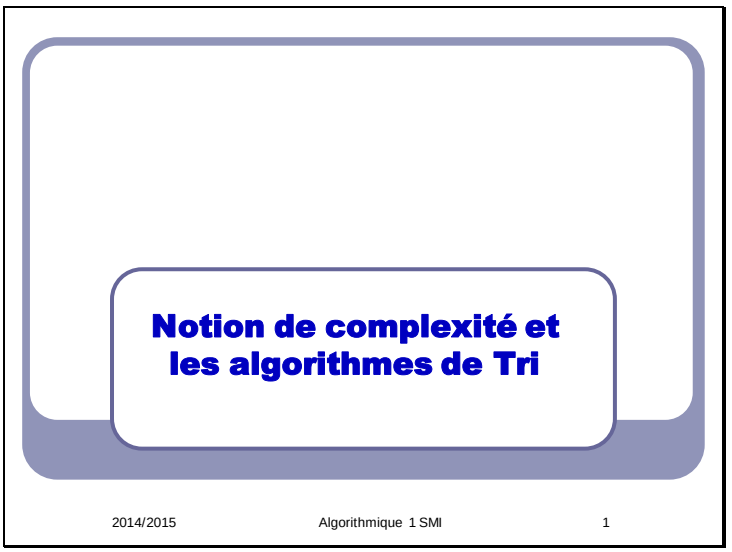
Contraintes du jeu

```
si pion-rouge alors
   si case-gauche est vide alors déplacement-gauche
   sinon si case-gauche-noire et case-suivante est vide alors
                                   déplacement-saut-gauche
sinon
   si pion-noir alors
     si case-à-droite est vide alors déplacement-droite
     sinon si case-droite-rouge et case-suivante est vide alors
                                   déplacement-saut-droite
   /*Déplacement ≈ échange de valeurs entre la case vide et
   la case du pion à jouer */
```


Programme principal

```
début
  répéter initPlateau(plateauJeu,nbPions);
     affichePlateau(plateauJeu,nbPions);
     suite←non finJeu(plateauJeu,nbPions,indVide);
     tant que (suite) faire
        jouerUnCoup(plateauJeu,nbPions,indVide);
        affichePlateau(plateauJeu,nbPions);
        suite ← non  finJeu(plateauJeu,nbPions,indVide);
     ftq
  tant que (rejouer)
Fin
```

Chap8 : L'introduction à la complexité des algorithmes et les tris



Notion de complexité

- L'exécution d'un algorithme sur un ordinateur consomme des ressources:
  - en temps de calcul : complexité temporelle
  - en espace-mémoire occupé : complexité en espace
- Seule la complexité temporelle sera considérée pour évaluer l'efficacité et la performance de nos programmes.

Notion de complexité

- Le temps d'exécution dépend de plusieurs facteurs :
  - Les données (trier 4 nombres ne peut être comparé au trie de 1000 nombres).
  - Le code généré par le compilateur (interpréteur).
  - La nature de la machine utilisée (mémoire, cache, multi-treading,…)
  - La complexité de l'algorithme.
- La complexité d'un algorithme permet de qualifier sa performance par rapport aux autres algorithmes.

Complexité d'un algorithme

- Si T(n) dénote le temps d'exécution d'un programme sur un ensemble de données de taille n alors :
- $T(n)=c.n^2$ (c est une constante) signifie que l'on estime à $c.n^2$ le nombre d'unités de temps nécessaires à un ordinateur pour exécuter le programme.
- Un algorithme "hors du possible" a une complexité temporelle et/ou en espace qui rend son exécution impossible.

  exemple: jeu d'échec par recherche exhaustive de tous les coups possibles

  10^{19} possibilités, 1 msec/poss. = 300 millions d'années

Complexité : exemple

- Écrire une fonction qui permet de retourner le plus grand diviseur d'un entier.

```
Fonction PGD1( n: entier) : entier
Variables i :entier
Debut
i ← n-1 ;
Tantque (n%i !=0)
    i ← i-1;
finTantque
Retourner( i)

Fin
```

```
Fonction PGD2( n: entier) : entier
Variables i :entier
Debut
   i ←2;
Tantque ((i<sqrt(n))&&(n%i !=0))
        i ← i+1;
 finTantque
  si(n%i == 0) alors retourner (n/i)
  sinon retourner (1)
  finsi
Fin
```

Pour un ordinateur qui effectue 10^6 tests par seconde et $n=10^{10}$ alors le temps requis par PGD1 est d'ordre 3 heures alors que celui requis par PGD2 est d'ordre 0.1 seconde

Complexité : notation en O

- La complexité est souvent définie en se basant sur le pire des cas ou sur la complexité moyenne. Cependant, cette dernière est plus délicate à calculer que celle dans le pire des cas.
- De façon général, on dit que T(n) est O(f(n)) si $\exists$ c et n0 telles que $\forall$ n≥n0, T(n) ≤ c.f(n). L'algorithme ayant T(n) comme temps d'exécution a une complexité d'ordre O(f(n))

$$\lim_{n \to +\infty} T(n)/f(n) <= c$$

Complexité : notation en O

- La complexité croit en fonction de la taille du problème
  - L'ordre utilisé est l'ordre de grandeur asymptotique.
  - Les complexités n et 2n+5 sont du même ordre de grandeur.
  - n et n^2 sont d'ordres différents.

Complexité : règles

- 1- Dans un polynôme, seul le terme de plus haut degré compte.
  - Exemple : $n^3+1006n^2+555n$ est $O(n^3)$
- 2- Une exponentielle l'emporte sur une puissance, et cette dernière sur un log.

  Exemple: 2^n+n^{100} est $O(2^n)$ et $300\lg(n)+2n$ est $O(n)$
- 3- Si T1(n) est O(f(n)) et T2(n) est O(g(n)) alors T1(n)+T2(n) est O(Max(f(n),g(n))) et T1(n).T2(n) est O(f(n).g(n))
- Les ordres de grandeur les plus utilisées :
  - O(log n), O(n), O(n log n), $O(n^k)$, $O(2^n)$

La complexité asymptotique

Supposons que l'on dispose de 7 algorithmes dont les complexités dans le pire des cas sont d'ordre de grandeur 1, $\log_2 n$, n, $n\lg_2 n$, n^2, n^3, 2^n et un ordinateur capable d'effectuer 10^6 opérations par seconde. Le tableau suivant montre l'écart entre ces algorithmes lorsque la taille des données croit :

Complexité	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
$N=10^2$	1µs	6.6µs	0.1ms	0.6ms	10ms	1s	4.10^{16}a
$N= 10^3$	1µs	9.9µs	1ms	9.9ms	1s	16.6mn	! (>10^{100})
$N=10^4$	1µs	13.3µs	10ms	0.1s	100s	11.5j	!
$N= 10^5$	1µs	16.6µs	0.1s	1.6s	2.7h	31.7a	!
$N= 10^6$	1µs	19.9µs	1s	19.9s	11.5j	31.710^3a	!

Tableaux : recherche d'un élément

- Pour effectuer la recherche d'un élément dans un tableau, deux méthodes de recherche sont considérées selon que le tableau est trié ou non :
  - La recherche séquentielle pour un tableau non trié
  - La recherche dichotomique pour un tableau trié

Méthode séquentielle

- Consiste à parcourir un tableau non trié à partir du début et s'arrêter dés qu'une première occurrence de l'élément sera trouvée.
- Le tableau sera parcouru du début à la fin si l'élément n'y figure pas.

Recherche séquentielle : algorithme

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

```
Variables i: entier, Trouve : booléen
...
i←0 ; Trouve ← Faux;
TantQue (i < N) ET (not Trouve)
      Si (T[i]=x) alors Trouve ← Vrai;
      Sinon  i←i+1;
      FinSi
FinTantQue
Si Trouve alors // c'est équivalent à écrire Si Trouve=Vrai alors
          écrire ("x est situé dans la "+i+ "eme position du tableau ");
Sinon     écrire ("x n'appartient pas au tableau");
FinSi
```

Recherche séquentielle : complexité

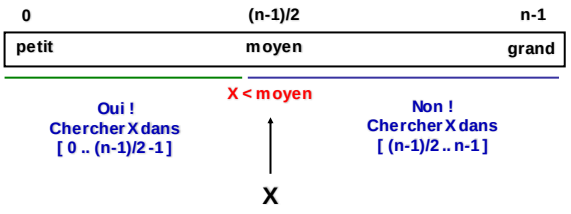
- Dans le pire des cas, on doit parcourir tout le tableau. Ainsi, la complexité est de l'ordre de O(n).
- Si le tableau est trié, la recherche séquentielle peut s'arrêter dés qu'on rencontre un élément du tableau strictement supérieur à l'élément recherché.
- Si tous les éléments sont plus petits que l'élément recherché, l'ensemble du tableau est parcouru. Ainsi la complexité reste d'ordre O(n).

Recherche dichotomique

- Dans le cas où le tableau est trié (ordonné), on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique (diviser pour régner).
- **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])
  - On continue le découpage jusqu'à un sous tableau de taille 1.

Recherche dichotomique

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



Recherche dichotomique : algorithme

```
inf←0 ; sup←N-1;Trouve ← Faux;
TantQue (inf <=sup) ET (not Trouve)
      milieu←(inf+sup) div 2;
      Si (x<T[milieu]) alors sup←milieu-1;
      Sinon Si (x>T[milieu]) alors inf←milieu+1;
            Sinon Trouve ← Vrai;
            FinSi
      FinSi
FinTantQue
Si Trouve alors écrire ("x appartient au tableau");
Sinon      écrire ("x n'appartient pas au tableau");
FinSi
```


Recherche dichotomique : exemple

- Considérons le tableau T :

3	7	9	12	15	17	27	29	37

- Si la valeur cherché est 16 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 9 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

- A chaque itération, on divise les indices en 3 intervalles :
  - [inf, milieu-1] : Cas 1
  - Milieu : Cas 2
  - [milieu+1, sup] : Cas 3

  Cas 1 : milieu-inf ≤ (inf+sup)/2 - inf ≤ (sup-inf)/2
  Cas 2 : =(sup-inf)/2
  Cas 3 : sup-milieu ≤ sup -(inf+sup)/2 ≤ (sup-inf)/2

- On passe dans successivement à un intervalle dont le nombre d'éléments ≤ n/2, puis n/4, puis n/8, …
- A la fin, on obtient un intervalle réduit à 1 ou 2 éléments.

Recherche dichotomique : complexité

- Le nombre d'éléments à la k ième itération est : $(½)^{k-1}n >=2$;
- Donc $2^k \leq n$ soit $k \leq \log_2 n$
- Il y a au plus $\log_2 n$ itérations comportant 3 comparaisons chacune.
- La recherche dichotomique dans un tableau trié est d'ordre $O(\log_2 n)$.

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-échange
  - Le tri par insertion
  - Le tri rapide
  - …
- Nous verrons dans la suite les trois algorithmes de tri. Le tri sera effectué dans l'ordre croissant.

Tri par sélection-échange

- **Principe** : C'est d'aller chercher le plus petit élément du tableau pour le mettre en premier, puis de recommencer à partir du second, d'aller chercher le plus petit élément pour le mettre en second etc…

  Au i-ème passage, on sélectionne le plus petit élément parmi les positions i..n et on l'échange ensuite avec T[i].

Tri par sélection-échange

- **Exemple :**

9	6	2	8	5

  - **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 :

2	6	9	8	5

  - **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:
  - **Étape 3:**

2	5	9	8	6

2	5	6	8	9

Tri par sélection-échange : algorithme

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

```
Pour i← 0 à N-2                                   Fin à n-2 !
    indice_ppe ← i;
    Pour j ← i + 1 à N-1
        Si T[j] <T[indice_ppe] alors              Chercher l'indice du
            indice_ppe ← j;                       plus petit à partir de i.
        Finsi
    FinPour
    temp ← T[indice_ppe];
    T[indice_ppe] ← T[i];          Echange, même si i = indice_ppe.
    T[i] ← temp;
FinPour
```


Tri par sélection-échange : complexité

- On fait n-i fois, pour i de 0 à n-2 :
- Il y a donc un nombre de lectures qui vaut :

$$\sum_{i=0..n-2} (n-i) = n+(n-1)+\ldots+2 = n(n+1)/2 \approx 0\ (n^2)$$

Tri en complexité quadratique.

Tri par insertion

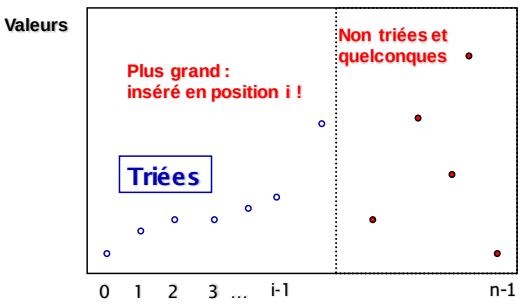
À la ième étape :

- Cette méthode de tri insère le ième élément T[i-1] à la bonne place parmi T[0], T[1]…T[i-2].
- Après l'étape i, tous les éléments entre les positions 0 à i-1 sont triés.
- Les éléments à partir de la position i ne sont pas triés.

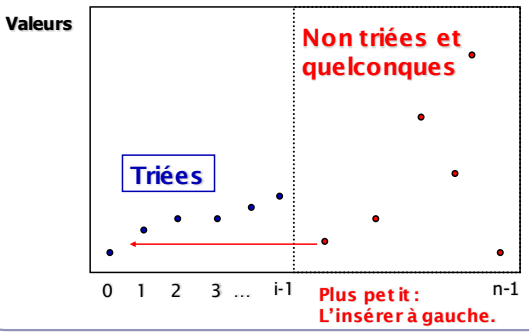
Pour insérer l'élément T[i-1] :

- Si T[i-1] ≥T[i-2] : insérer T[i-1] à la ième position !
- Si T[i-1] <T[i-2] : déplacer T[i-1] vers le début du tableau jusqu'à la position j ≤ i-1 telle que T[i-1] ≥T[j-1] et l'insérer à en position j.

Tri par insertion



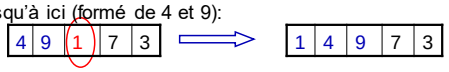
Tri par insertion



Tri par insertion : exemple

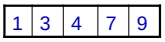
- **Étape 1:** on commence à partir du 2 ième élément du tableau (élément 4). On cherche à l'insérer à la bonne position par rapport au sous tableau déjà trié (formé de l'élément 9) :

- **Étape 2:** on considère l'élément suivant (1) et on cherche à l'insérer dans une bonne position par rapport au sous tableau trié jusqu'à ici (formé de 4 et 9):



- **Étape 3:**

- **Étape 4:**



Tri par insertion : algorithme

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

```
Pour i allant de 1 à N-1                          le premier élément est
    decaler ←vraie; j ←i;                         forcément à sa place
    Tantque ((j >0) et (decaler))
        Si T[j] <T[j-1] alors temp ← T[j];        On échange aussi
                               T[j] ← T[j-1];     longtemps que cela
                               T[j-1] ← temp;     est possible
        sinon decaler ←faux;
        Finsi
        j←j-1;
    FinTantque
FinPour
```


Tri par insertion : la complexité

- On fait, pour i de 1 à n-1 :
- Jusqu'à i échanges au maximum (peut-être moins).
- Le nombre d'échanges peut donc atteindre :

$$\sum_{i=1..n-1} i = 0\ (n^2)$$

Tri en complexité quadratique.

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 choisit un élément du tableau qu'on appelle pivot

Tri rapide

- **Description du tri rapide :**
  - **2)** Puis**,** on construit le sous tableau T1 comprenant tous les éléments inférieurs ou égaux au pivot. Et un sous-tableau T2 comprenant tous les éléments supérieurs au 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 seul élément

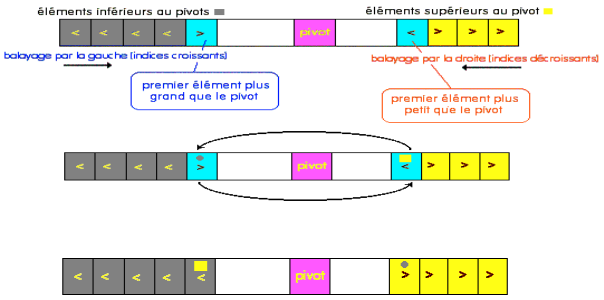
Tri rapide

- 0) Choix arbitraire du pivot que l'on cherche à placer. Balyage du tableau dans les deux sens.
- 1) balayage par la gauche, on s'arrête dès que l'on rencontre un élément dont la valeur est plus grande que le pivot.
- 2) balayage par la droite, on s'arrête dès que l'on rencontre un élément dont la valeur est plus petite que le pivot.
- 3) on procède à l'échange des deux éléments mal placés dans chacun des sous tableaux.

Tri rapide

- 4) on continue le balayage par la gauche et par la droite tant que les éléments sont bien placés, en échangeant à chaque fois les éléments mal placés.
- 5) la construction des deux sous tableaux est terminée dés que l'on atteint (ou on dépasse) le pivot.

Tri rapide



Tri rapide : exemple

- T = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]
  Avec un pivot = 16 (la dernière valeur), on obtient deux sous-tableaux :
  T1 = [4, 14, 3, 2]
  T2 = [42, 23, 45, 18, 38]
- **À cette étape l'arrangement de T est :**
  L = L1 + pivot + L2 = **[4, 14, 3, 2, 16, 42, 23, 45, 18, 38]**
- Remarquer que le pivot 16 est placé au bon endroit dans T.
- En appliquant la même démarche aux deux sous-tableaux T1 (pivot=2) et T2 (pivot=38)
  - T11=[ ] liste vide
  - T12=[**4,14, 3**]
  - T1=T11 + pivot + T12 = [**2,4, 14, 3**]

  T21=[**18, 23**]
  T22=[**45, 42**]
  T2=T21 + pivot + T22 = [**18, 23, 38, 45,42**]
- **À cette étape T=[(2, 4, 14, 3), 16, (18, 23, 38, 45, 42)]**
- **Etc…**

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
```


Procédure Tri rapide

- 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 T[q+1..r].
- L'indice q est calculé pendant la procédure de partitionnement.
- Les deux sous-tableaux T[p..q-1] et T[q+1..r] sont triés par des appels récursifs.

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
Debut
pivot← T[p]; i←p+1; j ← r;
TantQue (i<=j) faire
    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 : la complexité

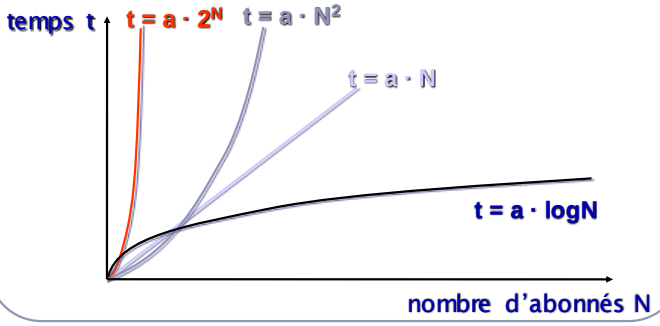
- Le tri rapide a une complexité moyenne d'ordre $O(n \log_2 n)$.
- Dans le pire des cas, le tri rapide reste d'ordre $O(n^2)$.
- Le choix du pivot influence largement les performances du tri rapide.
- Le pire des cas correspond au cas où le pivot est à chaque choix le plus petit élément du tableau (tableau déjà trié).
- différentes versions du tri rapide sont proposées 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.

Tri : Analyse de complexité

- Tri insertion ou Tri sélection sont d'ordre $O(N^2)$
  - Si $N=10^6$ alors $N^2 = 10^{12}$
  - Et si on peut effectuer 10^6 opérations par seconde alors l'algorithme exige 11,5 jours
- Tri rapide est d'ordre $O(N\log_2 N)$
  - Si $N=10^6$ alors $N\log_2 N = 6N$
  - Et si on peut effectuer 10^6 opérations par seconde alors l'algorithme exige 6 secondes

Introduction à la complexité

Annuaire avec noms et coordonnées



Algorithme en o(n) ~ 10 000 secondes ~ 3 heures recherche d'un nom
o(n^2) ~ 10^8 secondes ~ 3000 heures : tri par ordre alphabétique bourrin

o(2^n) plusieurs siècles
o(Log N) ~ 4 secondes
o(N log N) ~ 12 heures tri par ordre alphabétique “quick sort”

Enregistrements

- Les langages de programmation offrent d’autres types de données appelés enregistrements.
- Un enregistrement est un regroupement de données qui doivent être considérés ensemble.
- Exemple: les fiches d’étudiants. Chaque fiche est caractérisée par : un nom et prénom, numéro d’inscription, ensemble de notes…
- En pseudo-code : enregistrement FicheEtudiant

```
Debut nom, prenom : chaine de caractères
      numero : entier
      tableau notes[10] : réel
Fin
```


Enregistrements

- Un enregistrement est un type comme les autres types.
- Ainsi la déclaration suivante :

  f, g : FicheEtudiant

  définit deux variables f et g enregistrements de type FicheEtudiant

- L’enregistrement FicheEtudiant contient plusieurs parties (champs), on y accède par leur nom précédé d’un point "." :
- f.nom désigne le champ (de type chaine) nom de la fiche f.
- f.notes[i] désigne le champ (de type réel) notes[i] de la fiche f.

Enregistrements

- Pour définir les champs d’un enregistrement, on écrit :

```
f: FicheEtudiant;
f.nom ← "XXXXX";
f.prenom ← "YYYYY" ;
f.numero ← 1256;
f.notes[2] ← 12.5;
```

- Les affectations entre enregistrement se font champ par champ.

Enregistrements : remarques

- La notion de **Classe (beaucoup** plus riche) dans les langages à Objets remplace avantageusement la notion d'enregistrement.
- Un champ peut être de type élémentaire ou de type enregistrement.
- Il est possible d'imbriquer sans limitation des enregistrements les uns dans les autres.

Enregistrements : remarques

- Exemple :

```
Enregistrement  Adresse
                    numero: Entier;
                    codePostal: Entier;
                    rue,ville: ChaîneDeCharactères;
                Fin;
Enregistrement  Personne
                    nom,prenom: ChaîneDeCharactères;
                    age : Entier ;
                adresse : Adresse;
Fin;
```


Utilisation des enregistrements

```
Procedure affiche(FicheEtudiant  v)
debut
  ecrire("No:",v.numero,  "-",v.prenom)
  Pour i allant  de 0 à v.notes.taille()
     écrire(v.notes[i],  " ")
  FinPour
finProcedure
```

- Enregistrement Complexe

```
Debut re : réel
      im: réel
Fin
```

Enregistrements : exemple

```
Fonction add( z1, z2 :Complexe par valeur)  : Complexe
  Debut Variable z: Complexe
          z.re=z1.re+z2.re;
          z.im=z1.im+z2.im;
         retourne(z)
  FinFonction
Programme principale
Variables   u, v, w: Complexe
          a, b, c, d : réel
Debut ecrire("Entrez 4 valeurs réelles :");
        lire(a,b,c,d);
        u.re ← a;          u.im ← b;       v.re ← c;          v.im ← d;
        ww ← add(u,v);
        ecrire("Somme( ,) = :", w.re,w.im);
Fin
```


Structures en C

- Déclaration :

```
struct personne {
    char nom[20];
    char prenom[20];
     int  no_employe;
}
```

- Ce type de structure est utilisé pour déclarer des variables de la manière suivante : `struct personne p1, p2;`
- Accès aux membres : `p1.nom="XAAA";p2.no_employe=20;`
- Initialisation : `struct personne p={"AAAA", "BBBB", 5644};`
- Tableau de structure : `struct personne T[100];`