Algorithmique et Programmation

Mme Soumia ZITI-LABRIM
s.ziti@fsr.ac.ma
Université Mohamed V
Faculté des Sciences
Département Informatique

Plan

- Introduction
- Eléments de base
- Structures conditionnelles
- Structures itératives
- Tableaux
- Sous algorithme
- Fichiers
- Complexité
- Algorithmes de tris

Introduction

But de l'enseignement

Obtenir de la «machine» qu'elle effectue un travail à notre place

Problème: expliquer à la «machine» comment elle doit s'y prendre

Mais... comment le lui dire ou le lui apprendre afin qu'il fasse le travail aussi bien que nous, voir mieux que nous?

Objectifs

- Résoudre des problèmes «comme» une machine
- Savoir ***expliciter*** son raisonnement
- Savoir ***formaliser*** son raisonnement
- Concevoir et écrire des ***algorithmes*** (séquence d'instructions qui décrit comment résoudre un problème particulier)

Introduction

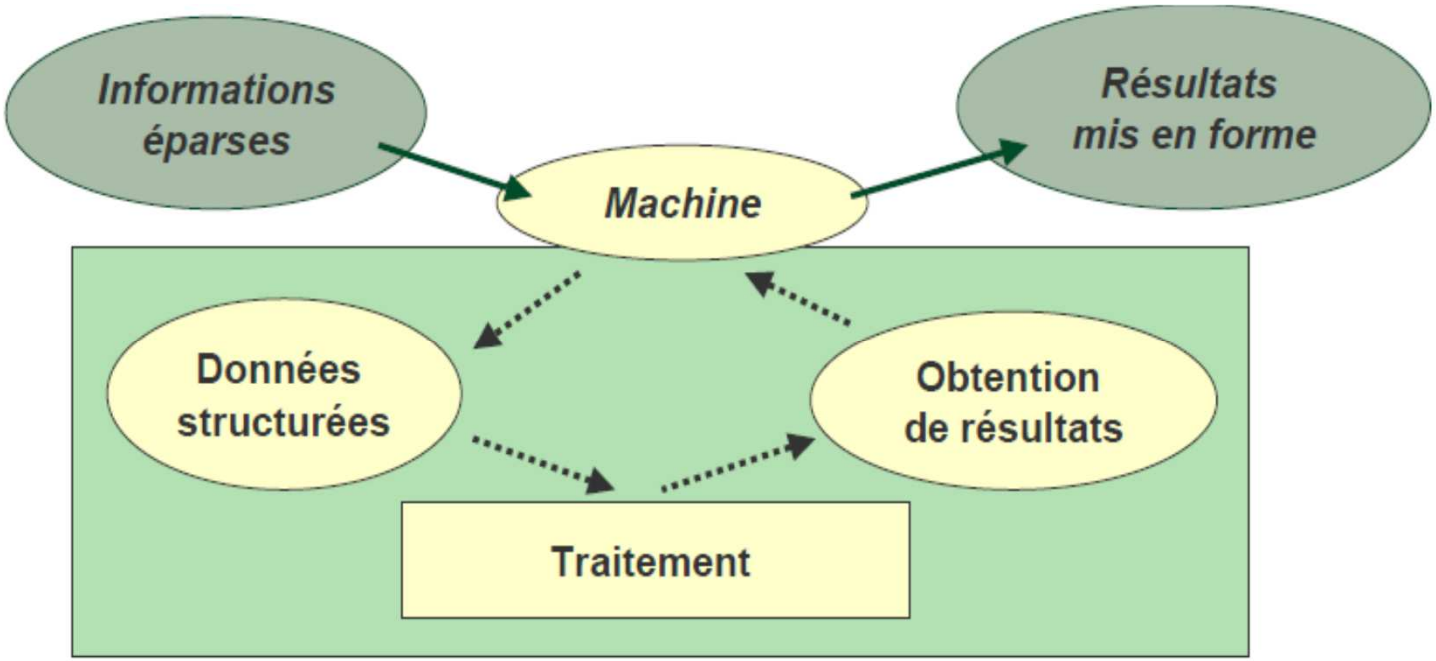
Algorithme

- Selon le **Petit Robert** : "ensemble des **règles** opératoires propres à un **calcul**."
- **Un peu plus précisément** : Une **séquence** de pas de **calcul** qui prend un ensemble de valeurs comme **entrée** et produit un ensemble de valeurs comme **sortie**.
- Un algorithme **résout** toujours un **problème** de calcul. L'énoncé du problème spécifie la **relation E/S** souhaitée.

Introduction

Algorithme

Un **algorithme**, traduit dans un langage compréhensible par l'ordinateur (ou langage de programmation, ici le C), donne un **programme**, qui peut ensuite être exécuté, pour effectuer le **traitement** souhaité.



Introduction

Algorithme

Savoir expliquer comment faire un travail sans la moindre ambiguïté

- **Langage simple** : des instructions séquentielle
- **Suite finie d'actions** à entreprendre en respectant une chronologie imposée

Un algorithme est indépendant de

- Le ***langage*** dans lequel il est implanté,
- La ***machine*** qui exécutera le programme correspondant.

Introduction

Structure d’un algorithme

Un algorithme doit être lisible et compréhensible par plusieurs personnes.

Algorithme : Nom d’Algorithme

Données : Les entrées de l’algorithme

Résultats : Les sorties de l’algorithme

Déclarations : Variables, constantes…

Début

Ensemble d’instructions ;

Fin

Introduction

Les étapes d'un algorithme

- **Préparation du traitement**
  - Données nécessaires à la résolution du problème
- **Traitement**
  - Résolution pas à pas, après décomposition en sous-problèmes si nécessaire
- **Edition des résultats**
  - impression à l'écran, dans un fichier, etc.

Introduction

Exemple :

Algorithme CalculeInverse

Variables Nombre, Inverse: entiers *{déclarations: réservation d'espace-mémoire}*

Début

{préparation du traitement}

Ecrire("Quel nombre voulez-vous élever au carré?")
Lire(Nombre)

{traitement : calcul de l'inverse}

Inverse ←1 / Nombre

{présentation du résultat}

Ecrire("L'inverse de ", Nombre,"c'est ", Inverse)

fin.

Introduction

Les problèmes fondamentaux

- **Complexité**
  - En combien de temps un algorithme va -t-il atteindre le résultat escompté?
  - De quel espace a-t-il besoin?
- **Calculabilité**
  - Existe-t-il des tâches pour lesquelles il n'existe aucun algorithme ?
  - Etant donnée une tâche, peut-on dire s'il existe un algorithme qui la résolve ?
- **Correction**
  - Peut-on être sûr qu'un algorithme réponde au problème pour lequel il a été conçu?

Introduction

Logique propositionnelle :

La logique : une façon de formaliser notre raisonnement

La logique propositionnelle: modèle mathématique qui nous permet de raisonner sur la nature vraie ou fausse des expressions logiques

Proposition: **expression** qui peut prendre la valeur **VRAI** ou **FAUX**

Exemple

x > y

1+1=2

1+1=1

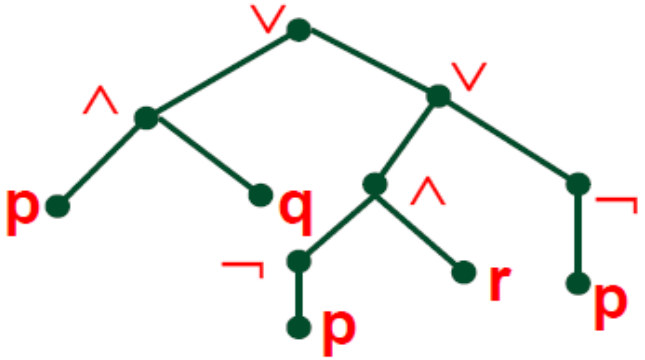
Introduction

Eléments de logique propositionnelle

- **Formule :** expression logique composée de variables propositionnelles et de connecteurs logiques
- **Variable propositionnelle :** une proposition considérée comme indécomposable
- **Connecteurs logiques:** négation ***non*** ,¬ ; implication ⇒ ; disjonction ***ou,*** ∨ ; conjonction ***et*** , ∧
- **Exemple :** p et q variables propositionnelles

(p ∧q) ∨((¬p ∧r) ∨¬p)

Par un **arbre syntaxique** :



En utilisant la notation **préfixée** :

∨∧p q ∨∧¬p r ¬p

En utilisant la notation **postfixée**:

p q ∧p ¬r ∧p ¬∨∨

Introduction

Tables de vérité

Représentation des valeurs de vérité associées à une expression logique

p et q : variables propositionnelles

Négation

p	¬ p
V	F
F	V

Conjonction

p	q	p∧q
V	V	V
V	F	F
F	V	F
F	F	F

Disjonction

p	q	p∨q
V	V	V
V	F	V
F	V	V
F	F	F

Implication

p	q	p→q
V	V	V
V	F	F
F	V	V
F	F	V

Introduction

Equivalences classiques

- **Commutativité**
  - **p ∧q** équivalent à **q ∧p**
  - **p ∨q** équivalent à **q ∨p**
- **Associativité**
  - **p ∧(q∧r)** équivalent à **(p ∧q) ∧r**
  - **p ∨(q ∨r)** équivalent à **(p ∨q) ∨r**
- **Distributivité**
  - **p ∧(q ∨r)** équivalent à **(p ∧q)∨(p∧r)**
  - **p ∨(q ∧r)** équivalent à **(p ∨q)∧(p∨r)**

Introduction

Lois de Morgan

¬(p ∧q) équivalent à **(¬p) ∨(¬q)**
¬(p ∨q) équivalent à **(¬p) ∧(¬q)**

p	q	p ∧q	¬(p ∧q)	¬p	¬q	¬p ∨ ¬q
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

Introduction

Formules :

Les tautologies : vraies pour toute assignation de valeurs de vérité aux variables.

Exemple: p∨¬p

p	¬ p	p ∨ ¬ p
0	1	1
1	0	1

Les formules contradictoires : fausses pour toute assignation de valeurs de vérité aux variables.

Exemple: p∧¬p

p	¬ p	p ∧ ¬ p
0	1	0
1	0	0

Introduction

Formules

Les formules équivalentes: même valeur de vérité pour toute assignation de la même valeur de vérité aux variables.

Exemples:
p ⇒q est équivalent à¬p ∨q
p ⇒q est équivalent à¬q ⇒¬p

p	q	p ⇒ q	¬ p	q	¬ p ∨ q
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Introduction

Applications à l'algorithmique

Interpréter(et bien comprendre!) l'arrêt des itérations

à la sortie d'une boucle.

tant que<condition> **faire**

À la sortie :**non**(<condition>)*est vrai*

donc si condition= p **et** q

À la sortie : **non**(p **et** q)

c'est a dire **non** p **ou non** q

Exemple:

avec **<condition>** égal à: val ≠STOP **et** nbVal< MAX

non(<condition>) égal à: val =STOP **ou**nbVal≥MAX

Introduction

Applications à l'algorithmique

- **Simplifier une écriture par substitution d'une formule équivalente**

si (Age = "Mineur"
ou (non (Age = "Mineur") et non (Fisc = "Imposable"))) alors...

Equivalent à :

si (Age = "Mineur" ou non (Fisc = "Imposable")) alors...

- **Vérifier la validité d'une condition**

si Valeur< 10 et Valeur >100 alors… *cas improbable*

- **Ecrire la négation d'une condition**

si on veut P et Q et R :

répéter …. tant que **non**P **ou non**Q **ou non**R **ou**…

Eléments de base

Variable

- Elle peuvent **stocker** des chiffres, des nombres, des caractères des chaîne de caractères..., dont **la valeur peut être modifiée** au cours de l'exécution de l'algorithme
- Une variable est une entité qui contient une information, elle possède :
  un nom ou **identifiant,** une **valeur** et un **type** qui caractérise l'ensemble des valeurs que peut prendre la variable
  - L'ensemble des variables est stocké dans la mémoire de l'ordinateur

Déclaration

Variable : <liste d'identificateurs1> : typeVaraible1
<liste d'identificateurs2> : typeVariable2

Exemple

Variable A, B : entier
d : réel

Eléments de base

Entier

- C'est le type qui représente des nombres **entiers relatifs** (**int**, **integer**)
- Il peut être codé en entier **simple** sur deux octets ou **long** sur quatre octets
- Il peut être représenté en **décimal** (0, - 55…), en **hexadécimal** (10h, 4Ah…) ou en **binaire** (% 01001, % 1110)

Réel

- C'est le type qui représente des nombres **réels** **(float)**
- Il peut être codé en réel **simple** sur 4 octets ou **double** sur 8 octets
- Il peut être représenté en forme **simple** (2.5, -2.0…) ou **exponentielle** (2.1 e^{4}, -6,98 E^{-2}…)

Eléments de base

Caractère

- Il est représenté en code **ASCII**
- Il permet d'avoir une relation **d'ordre**
- **Exemple** 'A' < 'a' car en ASCII 65<97 et 'A'<'Z' car en ASCII 65<90

Chaîne de caractère

- Elle représente un **tableau de caractères**
- Plusieurs **fonctions prédéfinies** : **Longueur(S**) donne la longueur de S

Booléen

- Il présente les deux valeurs **Vrai** et **Faux** (**True** and **False** ou **1** et **0**)

Eléments de base

Constante

- Elle peuvent stocker des chiffres, des nombres, des caractères des chaîne de caractères..., dont la valeur ne peut être modifiée au cours de l'exécution de l'algorithme

- **Déclaration**

Constante : <NomVariable> ← <ValeurVariable> : TypeVariable

- **Exemple**

Constante A←1 : entier

Eléments de base

Opérateurs

- Un **opérateur** est un symbole d'opération qui permet d'agir sur des variables ou de faire des "calculs"
- Une **opérande** est une entité (variable, constante ou expression) utilisée par un opérateur
- Une **expression** est une combinaison d'opérateur(s) et d'opérande(s), elle est évaluée durant l'exécution de l'algorithme, et possède une valeur (son interprétation) et un type
- **Exemple**
  - Dans l'expression **a + b**, a et b sont des opérandes et + l'opérateur
  - Dans l'expression **c= a* b** : c, a, b et a*b sont des opérandes et = et * sont des opérateurs
  - Si par exemple a et b sont des entiers, l'expression a + b, a*b et c sont aussi des entier

Eléments de base

Types opérateurs

- Un opérateur est **unaire** (non) ou **binaire** (+)
- Un opérateur est associé à ***un type*** et ne peut être utilisé qu'avec des données de *ce* type

Arithmétiques
Addition : **+** (ou concaténation)
Soustraction : **-**
Multiplication : *****
Division : ***/***
Division entière : **DIV**
Puissance : ↑
Reste de DIV : **MOD**

Comparaisons
Inférieur : **<**
Inférieur ou égale : **<=**
Supérieur : **>**
Supérieur ou égale : **>=**
Différent : **!=**
Egale : **=**

Logiques
Conjonction : **ET**
Disjonction : **OU**
Disjonction exclusive : **OUX**
Négation : **NON**
Décalage à droit : **>>**
Décalage à gauche : **<<**

Eléments de base

Priorité des Opérateurs

- En **arithmétique** les opérateurs * et / sont prioritaires sur + et -
- Pour les **booléens**, la priorité des opérateurs est **non**, **et**, **ouExclusif** et **ou**

Opérateur d'affectation

- Il permet d'affecter une valeur de l'opérande droit à une variable (opérande gauche), il est représenté par : ←

<Identificateur> ← <expression> || <constante> || <identificateur>

Exemple:

nom ← "Venus "

val1 ← val2

val ← val ×2

Eléments de base

Entrée\Sortie

- Un algorithme peut avoir des **interactions** avec l'utilisateur et **communiquer** avec lui dans les **deux sens**, les **sorties** sont des envois de messages a l'utilisateur, les **entrées** sont des informations fournies par l'utilisateur.
- Il peut demander à l'utilisateur de **saisir** une information afin de la stocker dans une variable et peut **afficher** un résultat (du texte ou le contenu d'une variable)

Eléments de base

Instruction d'écriture (Sortie)

Elle permet la restitution de résultats sur le périphérique de sortie (en général l'écran)

Syntaxe : **écrire(liste d'expressions)**

- Cette instruction réalise simplement l'affichage des valeurs des expressions décrites dans la liste.
- Ces instructions peuvent être simplement des variables ayant des valeurs ou même des nombres ou des commentaires écrits sous forme de chaînes de caractères.

Exemple :

écrire(x, y+2, "bonjour")

Eléments de base

Instruction lecture (Entrée)

L'instruction de prise de données sur le périphérique d'entrée (en général le clavier)

Syntaxe : **lire(liste de variables)**

- L'exécution de cette instruction consiste à affecter une valeur à la variable en prenant cette valeur sur le périphérique d'entrée

Exemple :

Lire(x, y, A)

Eléments de base

Exemple

Cet algorithme demande a l'utilisateur de **saisir** une valeur numérique, ensuite il **affiche** la valeur saisie, puis la même valeur incrémentée de 1.

Algorithme : Affichage incrément
variables :
    a, b : entier
DEBUT
    écrire("Saisissez une valeur numérique")
    lire(a)
    b ← a + 1
    écrire("Vous avez saisi la valeur ", a, ". ")
    écrire(a, "+ 1 = ", b)
FIN

Structure conditionnelle

La structure Si

L'instruction si alors sinon permet de conditionner l'exécution d'un algorithme à la valeur d'une expression booléenne.

Syntaxe :

si ***<expression booléenne>*** **alors**
<suite d'instructions exécutées si l'expression est vrai>
sinon
<suite d'instructions exécutées si l'expression est fausse>
finsi

La deuxième partie de l'instruction est optionnelle, on peut avoir :

si ***<expression booléenne>*** **alors**
<suite d'instructions exécutées si l'expression est vrai>
finsi

Structure conditionnelle

Structure Si

Exemple

Algorithme : Valeur Absolue

Données : La valeur à calculer

Résultat : La valeur Absolue

début

si valeur ≥ 0 alors

valeurabsolue ← valeur

sinon

valeurabsolue ← valeur * -1

finsi

fin

Structure conditionnelle

Structure de choix multiple

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

```
si abréviation = "M" alors
        écrire( "Monsieur" )
Sinon
        si abréviation = "Mme" alors
                écrire("Madame")
        sinon
                si abréviation = "Mlle" alors
                        écrire( "Mademoiselle" )
                sinon
                        écrire( "Monsieur, Madame " )
                fsi
        fsi
fsi
```

On peut remplacer cette suite de si par l'instruction **Selon**

Structure conditionnelle

Structure de choix multiple

Syntaxe

```
selon <identificateur : V> faire
        V1 : instructions 1
        V2 : instructions 2
                ...
        Vn : instructions n

        [autres: instructions]
finSelon
```

- V1,. . . ,Vn sont des **constantes** de type **scalaire** (entier, réel, caractère …)
- **instructions i** est exécutée si **V = Vi** (on quitte ensuite le **selon**)
- **instruction autre** est exécutée si quelque soit i, **V ≠ Vi**

Structure conditionnelle

Structure de choix multiple

Exemple

```
selon abréviation faire
        "M" : écrire( " Monsieur " )
        "Mme" : écrire( " Madame " )
        "Mlle" : écrire( " Mademoiselle " )
        autres: écrire( " Monsieur, Madame " )
finSelon
```

Structure itératives

- Un algorithme peut **répéter** le même traitement plusieurs fois, avec éventuellement quelques variantes.
- Dans certain cas, Il est impossible de savoir à l'avance **combien de fois** la même instruction doit être décrite.
- Utilisation des instructions en **boucle** qui répètent plusieurs fois une même instruction.
- **Deux formes existent** : la première, si le nombre de répétitions est **connu** avant l'exécution de l'instruction de répétition (**Pour**), la seconde s'il **n'est pas connu** (**Tant que** et **répéter… jusqu'à**). L'exécution de la liste des instructions se nomme **itération**.

Structure itératives

Répétition inconditionnelle

- Il est fréquent que le nombre de répétitions soit **connu** à l'avance, et que l'on ait besoin d'utiliser le numéro de **l'itération** afin d'effectuer des **calculs** ou des **tests**.

Syntaxe de Pour :

Pour <variable> de ValInit à ValFin [par <pas>] faire
liste d'instructions
FinPour

- La variable prend **successivement** toutes les valeurs entières entre valeur initiale et valeur finale. Pour chaque valeur, la liste des instructions est exécutée.
- La valeur utilisée pour énumérer les itérations est appelée **valeur** d'itération, **indice** d'itération ou **compteur**. L'incrémentation par 1 de la variable est **implicite**.

Structure itératives

Répétition conditionnelle

- Dans beaucoup de cas, on souhaite répéter une instruction tant qu'une certaine **condition est remplie**, alors qu'il est à priori **impossible** de savoir à l'avance au bout de combien d'itérations cette condition cessera d'être satisfaite.

- Dans ce cas, on a deux possibilités :
  - la boucle **Tant que**
  - la boucle **Répéter jusqu'à**

Structure itératives

Boucle Tant que

Syntaxe :

tant que condition faire
liste d'instructions
FinTantQue

- Cette instruction a une **condition** de poursuite dont la valeur est de type **booléen** et une liste d'instructions qui est répétée si la valeur de la condition de poursuite est **vraie** : la liste d'instructions est répétée autant de fois que la condition de poursuite a la valeur est **vraie**.
- Etant donné que la condition est évaluée **avant** l'exécution des instructions à répéter, il est possible que celles-ci ne soient **jamais** exécutées.
- Il faut que la liste des instructions ait une **incidence sur la condition** afin qu'elle puisse être évaluée à **faux** et que la boucle se termine (Il faut toujours s'assurer que la condition devient fausse au bout d'un temps fini.)

Structure itératives

Boucle Tant que

Exemple

Un utilisateur peut construire des rectangles de taille quelconque, à condition que les largeurs qu'il saisit soient supérieures à 1 pixel.

Algorithme : saisirLargeurRectangle
Variable : largeur : entier
début
    écrire ("indiquez la largeur du rectangle :")
    lire(largeur)
    tant que largeur < 1 **faire**
        écrire ("erreur : indiquez une valeur > 0")
        écrire ("indiquez la largeur du rectangle :")
        lire(largeur)
    finTantQue
fin

Structure itératives

Boucle Répéter jusqu'à

Syntaxe :

Répéter

liste d'instructions

jusqu'à condition

La séquence d'instructions est exécutée **au moins une fois** et jusqu'à ce que l'expression soit **vraie**. Dès que la condition est vrai, la répétitivité s'arrête.

Structure itératives

Boucle Répéter jusqu'à

Exemple

Algorithme : saisirLargeurRectangle
Variable : largeur : entier
début

```
        répéter
                écrire ("indiquez la largeur du rectangle :")
                lire(largeur)
                si largeur < 1 alors
                        écrire ("erreur : indiquez une valeur > 0")
                finSi
        jusqu'à largeur >= 1
fin
```

Structure itératives

Comparaison Tant que e Répéter jusqu'à

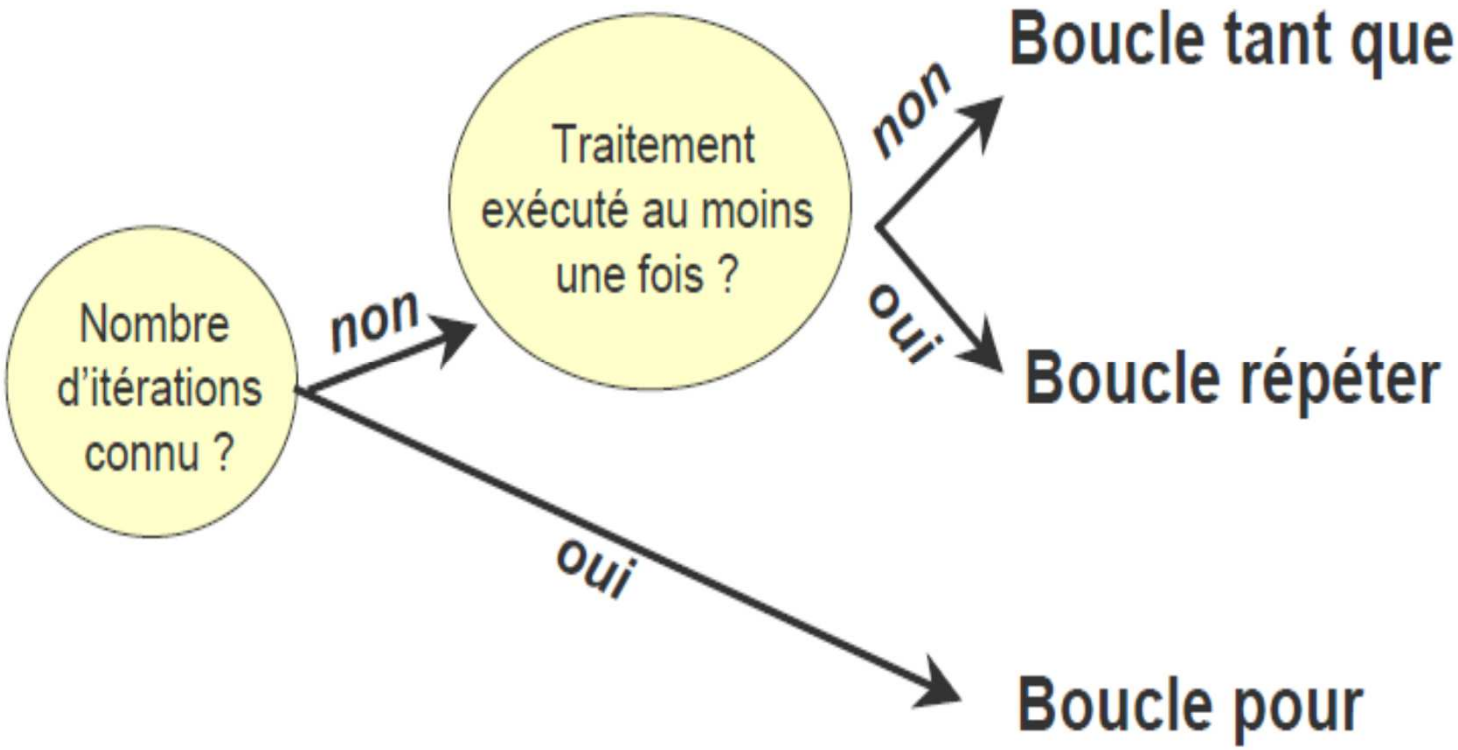
- La séquence d'instructions est exécuter **au moins une fois** dans la boucle Répéter jusqu'à, alors qu'elle peut **ne pas être exécuter** dans le cas du Tant que.

- la séquence d'instructions est exécuter si la condition est **vrai** pour le Tant que et si la condition est **fausse** pour le Répéter jusqu'à.

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

Structure itératives

Conclusion



Tableaux

- Lorsque les données sont nombreuses et de même type, afin d'éviter de multiplier le nombre des variables, on les regroupe dans un **tableau**

- Le type d'un tableau précise le type (**commun**) de tous les éléments.

- **Syntaxe** :

<nomTab> : tableau [borne_inf ... borne_sup] : <TypeTab>

- En général, nous choisirons toujours la valeur 0 pour la borne inférieure dans le but de faciliter la traduction de l'algorithme vers les autres langages (C, Java, ...). Pour un tableau de 10 entiers, on aura :

- **Exemple**

tabVal : tableau [0..9] : entier

Tableaux

Les tableaux à une dimension (vecteurs)

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. 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 instructions de **lecture**, **écriture** et **affectation** s'appliquent aux tableaux comme aux variables.

x ← Tab[0] (x=45)

Tab[6] ← 43

Tableaux

Les tableaux à deux dimensions (matrices)

- Lorsque les données sont nombreuses et de même nature, mais dépendent de deux critères différents, elles sont rangées dans un tableau à deux entrées.

	0	*1*	*2*	*3*	*4*
0	12	28	44	2	76
1	23	36	51	11	38
2	43	21	55	67	83

- Ce tableau a 3 lignes et 7 colonnes. Les éléments du tableau sont repérés par leur numéro de ligne et leur numéro de colonne.

Tab[1, 4] = 38

Tableaux

La variable **Tab[i, j]** s'appelle l'élément la ligne i et la colonne j (indice i et j du tableau Tab).

Tab[0, 0]	Tab[0, 1]	Tab[0, 2]	Tab[0, 3]	Tab[0, 4]
Tab[1, 0]	Tab[1, 1]	Tab[1, 2]	Tab[1, 3]	Tab[1, 4]
Tab[2, 0]	Tab[2, 1]	Tab[2, 2]	Tab[2, 3]	Tab[2, 4]

Syntaxe

<nomTab> : tableau [binf1 ... Bsup1; binf2 ... Bsup2] : <TypeTab>

Les tableaux peuvent avoir **n dimensions**.

Tableaux

Traitements opérant sur des tableaux

- **Créer** des tableaux
- **Ranger** des valeurs dans un tableau
- **Récupérer**, **consulter** des valeurs rangées dans un tableau
- **Rechercher** si une valeur est dans un tableau
- **Mettre à jour** des valeurs dans un tableau
- **Modifier** la façon dont les valeurs sont rangées dans un tableau (par exemple : les trier de différentes manières)
- Effectuer des **opérations entre tableaux** : comparaison de tableaux, multiplication,...

Tableaux

Parcours complet d'un tableau

.

- Les répétitions inconditionnelles sont le moyen le plus simple de parcourir complètement ou partiellement un tableau.

- **Syntaxe**

Pour une dimension :

Pour cpte **de** 0 à Taille(Tab)-1 [**par** 1] **faire**

Utiliser **Tab[cpte]** pour saisie, affichage, affectation, calcul…

FinPour

Pour deux dimension :

Pour ligne **de** 0 à nbLigne-1 [**par** 1] **faire**

Pour colonne **de** 0 à nbColonne-1 [**par** 1] **faire**

Utiliser **Tab[ligne,colonne]** pour saisie, affichage, …

FinPour

Tableaux

Exemple

Dans l'exemple suivant, le programme initialise un à un tous les éléments de deux tableau différents:

```
Algorithme : initialisation tableaux
Variable : i , j :entier
        Tab : tableau de [0…n-1] :entier
        Mat : tableau de [0…n-1 ; 0…n-1] :entier
        Constante : n ← 20 : entier
début
        pour i de 0 à n-1 faire
                tab[i] ← i
                pour j de 0 à n-1 faire
                        Mat[i , j] ← i*i
        finpour
fin
```

Sous algorithmes

Méthodologie de base:

1. Abstraire

Retarder le plus longtemps possible l'instant du **codage**

2. Décomposer

"...**diviser** chacune des difficultés que j'examinerai en autant de parties qu'il se pourrait et qu'il serait requis pour les **mieux** résoudre." ***Descartes***

3. Combiner

Résoudre le problème par **combinaison** d'abstractions

Sous algorithmes

Exemple

résoudre le problème suivant : Ecrire un programme qui **affiche en ordre croissant** les notes d'une promotion suivies de la **note la plus faible**, de la **note la plus élevée** et de **la moyenne**, revient à résoudre les problèmes suivants :

- **Remplir** un tableau de naturels avec des notes saisies par l'utilisateur
- **Afficher** un tableau de valeurs
- **Trier** un tableau de valeurs en ordre croissant
- Trouver la **plus petite** valeur d'un tableau
- Trouver la **plus grande** valeur d'un tableau
- Calculer la **moyenne** d'un tableau de valeurs
- 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

Sous algorithmes

Principe

- Un algorithme **appelle** un sous-algorithme : cet algorithme *passe "momentanément" le contrôle* de l'exécution du traitement au sous-algorithme.
- Un sous-algorithme est conçu pour faire un traitement bien **défini**, bien **délimité**, si possible *indépendamment* du contexte particulier de l'algorithme appelant.
- **Remarque :** un sous-algorithme peut en appeler un autre.
- En algorithmique il existe **deux types** de sous-programmes :
  - **Les fonctions**
  - **Les procédures**

Sous algorithmes

Fonction

- Une fonction est une **suite ordonnée** d'instructions, réalisant une certaine **tâche**. Elle admet **zéro**, **un** ou **plusieurs paramètres** et **retournant** toujours une valeur
- Une fonction joue le rôle d'une **expression**. Elle enrichit le jeu des expressions possibles.
- **Exemple :**

  Y= sin (X)

  Nom : sin

  Arguments ou paramètres : X

Sous algorithmes

Procédure

- Une procédure est une **suite ordonnée** d'instructions réalisant une certaine **tâche**. Elle admet **zéro**, **un** ou **plusieurs paramètres** et **ne renvoie pas** de résultat.
- Une procédure joue le rôle d'une **instruction**. Elle enrichit le jeu des instructions existantes.

- **Exemple :**

print (X, Y, Z)

Nom : print

Arguments ou paramètres : X, Y et Z

Fonction récursive

Caractéristiques

- Un programme est dit récursif s'il **s'appelle lui même**
- Un programme récursif est donc forcément une **fonction** ou une **procédure** (il doit pouvoir s'appeler)
- Il est impératif de prévoir une **condition d'arrêt** à la récursion, sinon le programme ne s'arrête jamais!
- Il faut toujours tester en **premier** la condition d'arrêt, et ensuite, si la condition n'est pas vérifiée lancer un appel récursif

Fonction récursive

- **Exemple *: la factorielle***

version itérative :

n ! = 1 x 2 x 3 x … x n

Algorithme : **Factorielle**

Données : n : **entier**

Résultat : le factoriel de n

Variable : i, r : **entier**

début

r ← 1

pour i **de** 2 **à** n **faire**

r ← r*i

retourne r

fin

version récursive:

1!=1 et pour n>1 n ! = n x (n-1)!

Algorithme : **Factorielle**

Données : n : **entier**

Résultat : le factoriel de n

début

Si n=1 **alors**

retourne 1

Sinon

retourne n***Factorielle**(n-1)

FinSi

fin

Fonction récursive

Conclusion

- 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 **dispendieuse** de ressources machine. Car il faut créer autant de variable **temporaires** que de " tours " de fonction en attente.

- Toute fonction récursives peut également être formulée en termes **itératifs** ! Donc, si elles facilitent la vie du programmeur, elle ne sont pas indispensable.

Fichiers

Besoin de fichiers

- Jusqu'à maintenant nous n'avions vu que des **Entrées/Sorties** (E/S) sous la forme **d'entrée standard** (usuellement le clavier, **lire**) et de **sortie standard** (usuellement l'écran, **écrire**)

- Ce type d'E/S atteint rapidement ses **limites**...
  - Utilisation des fichiers : **sauvegarde/restitution des données entre le programme et le disque dur.**

Fichiers

Définition

- **Un fichier (file)** est un **ensemble structuré de données** stocké en général sur un **support externe** (disquette, disque dur ou optique, ...). Ils servent à stocker des informations de manière **permanente**, entre deux exécutions d'un programme

- **Un fichier structuré** contient une suite d'enregistrements **homogènes**, qui regroupent le plus souvent **plusieurs composantes** (champs)

- **Un fichiers séquentiel** contient des enregistrements qui sont mémorisés **consécutivement** dans l'ordre de leur entrée et peuvent seulement être lus dans cet **ordre**. Si on a besoin d'un enregistrement précis dans un fichier séquentiel, il faut lire **tous** les enregistrements qui le **précèdent**, en commençant par le premier.

Fichiers

Types de fichiers

- **Les fichiers textes** : sont les fichiers dont le contenu représente uniquement **une suite de caractères** imprimables, d'espaces et de retours à la ligne (.txt,...). Ils peuvent être lus directement par un **éditeur de texte.**

- **Les fichiers binaires** : sont les fichiers qui ne sont pas **assimilables à des fichiers textes** (.exe, .mp3, .png,...). Ils ne peuvent pas être lus directement par un éditeur de texte .

Fichiers

Structures de fichiers

- **Structure délimitée** : Elle utilise un **caractère spécial**, appelé **caractère de délimitation**, qui permet de repérer quand finit un champ et quand commence le suivant. Il va de soi que ce caractère de délimitation doit être **strictement interdit** à l'intérieur de chaque champ, faute de quoi la structure devient proprement **illisible**.
  - "Fonfec"**;**"Sophie"**;**0142156487**;**"fonfec@yahoo.fr"

- **Structure à champs de largeur fixe** : les x **premiers caractères** de chaque ligne stockent le **premier champs**, les y suivants le second champs…..Il faut faire attention aux **longueurs des champs**
  - Fonfec Sophie 0142156487 fonfec@yahoo.fr

Fichiers

Types d'accès

- **L'accès séquentiel** : n'accéder à une information qu'en ayant au **préalable** examiné celle qui la précède. **Accès à un enregistrement par enregistrement**

- **L'accès direct** (ou **aléatoire**) : accéder **directement à l'enregistrement de son choix**, en précisant le **numéro** de cet enregistrement. Mais cela veut souvent dire une gestion **fastidieuse** des déplacements dans le fichier

- **L'accès indexé** : combiner la **rapidité** de l'accès direct et la **simplicité** de l'accès séquentiel. Il est particulièrement adapté au traitement des gros fichiers, comme les **bases de données** importantes.

Fichiers

Propriétés(fichier séquentiel)

- Un fichier est donné par son **nom** (et en cas de besoin le **chemin d'accès** sur le médium de stockage), il peut être **créé**, **lu** ou **modifié**

- Les fichiers se trouvent ou bien en état **d'écriture** (mettre dedans toutes les informations que l'on veut, mais les informations précédentes, si elles existent, seront **intégralement écrasées**), en état de **lecture** (récupérer les informations qu'il contient, sans les modifier en aucune manière) ou en **ajout** (ajouter de **nouvelles lignes** ou **nouveaux enregistrement**)

- A un moment donné, on peut uniquement accéder à un **seul enregistrement**; celui qui se trouve en face de la **tête** de lecture/écriture. Après chaque accès, la tête de lecture/écriture est **déplacée** derrière la donnée lue en dernier lieu.

Fichiers

Mécanisme

- Déclarer un fichier : **NomFichier : Fichier**
- Ouvrir le fichier en un mode spécifié : **Ouvrir(NomFichier) en mode**
- Tester que le fichier est bien ouvert **: Si NomFichier est Ouvert**
- Lire\ écrire dans le fichier ouvert :
  - **LireFichier(**NomFichier**,** valeurEnregistrement**)**
  - **EcrireFichier(**NomFichier**,** valeurEnregistrement**)**
- Fermer le fichier : **Fermer(NomFichier)**

Fichiers

Exemple

```
Variable Truc  : chaîne Caractère
         Fich : Fichier
Début
         Fich ← "Exemple.txt"
         Ouvrir(Fich)  en Lecture
         Tantque Non EOF(Fich)
                  LireFichier (Fich,Truc)
         FinTantQue
         Fermer(Fich)
         Truc ← "Fonfec";"Sophie";0142156487;"fonfec@yahoo.fr"
         Ouvrir(Fich)  en Ajout
         EcrireFichier (Fich, Truc)
         Fermer(Fich)
Fin
```

Complexité

Problématique :

- Mesurer **l'efficacité** d'un algorithme, ce qu'on appelle sa complexité
  - Prévoir son **temps** d'exécution
  - Estimer les **ressources** qu'il va mobiliser dans une machine lors de son exécution (place occupée en mémoire en particulier)
  - **Comparer** avec un autre qui fait le même traitement d'une autre façon, de manière à choisir le meilleur

- L'évaluation de la complexité peut se faire à plusieurs niveaux
  - Niveau purement **algorithmique**, par l'analyse et le calcul
  - Niveau du **programme**, par l'analyse et le calcul
  - Niveau de **l'exécution** du programme expérimentalement

Complexité

Complexité expérimentale

- Jusqu'aux années 70, seule **la mesure expérimentale** de la complexité d'un algorithme était (parfois) effectuée

- Cette évaluation expérimentale dépendait énormément des **machines** mais permettait de **comparer** l'efficacité de **différents algorithmes** si on les écrivait dans un **même langage** et qu'on les faisait tourner sur une **même machine**

- Si on les faisait tourner sur des **machines différentes**, il fallait évaluer la puissance des machines qui **dépend** du **matériel** mais aussi du **système d'exploitation** et **varie** en fonction des **traitements effectués** (calculs bruts, sur des entiers ou des réels, calculs liés à l'affichage, ...)

Complexité

Complexité en temps et en espace

- Plusieurs complexités peuvent être évaluées :
  - **Complexité en temps** : il s'agit de savoir combien de temps prendra l'exécution d'un algorithme
  - **Complexité en espace** : il s'agit de savoir combien d'espace mémoire occupera l'exécution de l'algorithme

- Souvent, si un algorithme permet de **gagner du temps de calcul**, il **occupe davantage de place en mémoire** (mais un peu d'astuce permet parfois de gagner sur les deux tableaux)

- Généralement, on s'intéresse essentiellement à la **complexité en temps** (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

Complexité

Complexité au pire

- La complexité n'est pas la même selon les déroulements du traitement
  - **Complexité au pire** : complexité **maximum**, dans le cas le plus défavorable (**borne supérieur** du temps d'exécution)
  - **Complexité en moyenne** : il s'agit de la **moyenne** des complexités obtenues selon les issues du traitement
  - **Complexité au mieux** : complexité **minimum**, dans le cas le plus favorable. En pratique, cette complexité n'est pas très utile

- Si on veut **comparer** les algorithmes **indépendamment** des implémentations et des machines, on ne peut comparer que la forme générale de la complexité, en particulier la **façon dont elle évolue selon la taille des données**

Complexité

Paramètre de la complexité

- La complexité d'un algorithme est calculée en fonction d'un **paramètre** par rapport auquel on veut calculer cette complexité

- Pour un algorithme qui opère sur une **structure de données** (tableau), la complexité est généralement exprimée en fonction d'une **dimension** de la structure.

- Si l'algorithme prend en entrée une **structure à plusieurs dimensions** il faut préciser en fonction de **quelle dimension** on calcule la complexité (l'algorithme peut avoir des complexités **différentes**)

- Pour un algorithme qui opère sur un **nombre**, la complexité est généralement exprimée en fonction de **la valeur** du nombre

Complexité

Complexité approchée

Calculer la complexité de **façon exacte n**'est pas **raisonnable** (quantité d'instructions) et n'est pas **utile** (comparaison des algorithmes)

- **Première approximation** : considérer souvent que la **complexité au pire**
- **Deuxième approximation** : calculer la **forme générale** de la complexité, qui indique la façon dont elle évolue en **fonction d'un paramètre**
- **Troisième approximation** : étudier **comportement asymptotique** de la complexité, quand la valeur du paramètre devient “assez” **grande**

Complexité

Notation Landau

- **g est dominée asymptotiquement par f** : **f(n) = O(g(n)) si ∃ c > 0 et n_o > 0 tels que ∀ n ≥ n_o f(n) ≤ c. g(n)**
- **f domine asymptotiquement g** : **f(n) = Ω(g(n)) si ∃ c > 0 et n_o > 0 tels que ∀ n ≥ no f(n) ≥ C. g(n)**
- **f est asymptotiquement équivalente à g** : **f(n) = Θ(g(n)) si ∃ c1 et c2 > 0 et no > 0 tels que ∀ n ≥ no c1.g(n) ≤ f(n) ≤ c2. g(n)**

Complexité

Exemples

- $x = O(x^2)$ car pour $x>1$, $x<x^2$
- $100*x = O(x^2)$ car pour $x>100$, $x<x^2$
- $x^2 = O(x^3)$ car pour $x>1$, $x^2<x^3$
- $\ln(x) = O(x)$ car pour $x>0$, $\ln(x)<x$
- $\forall i>0$, $x^i = O(e^x)$ car pour x tel que $x/\ln(x)>i$, $x^i<e^x$
- $O(x) = O(x2)$ mais $O(x2) \neq O(x)$
- Si $f(x) = O(1)$ alors f est majorée
- Si $f(x) = \Omega(1)$ f est minorée
- $6*2^x + x^2 = O(2^x)$
- $10n^2+4n+2= \Omega(n^2)$
- $6*2^n + n^2 = \Omega(n^2)$

Complexité

Types de complexité

- Complexité **constante** en O(1)
- Complexité **logarithmique** en O(log(n))
- Complexité **linéaire** en O(n)
- Complexité **quasi-linéaire** en O(nlog(n))
- Complexité **quadratique** en O(n2) :
- Complexité **polynomiale** en $O(n^i)$
- Complexité **exponentielle** en $O(i^n)$
- Complexité **factorielle** en O(n!)

Complexité

Exemple

Calcul de la valeur d'un polynôme en un point

1. p ← a[0]
2. **pour i ←0 à n-1 faire** *{puissance(a, n) retourne a^n}*
3. xpi ← puissance (x , i)
4. p ← p + a[i]* xpi

fpour

Nombre de multiplications

en 3 --> 1+2+3+...+ (n-1) = (n-l)n/2

en4 --> n

Nombre d'additions en 4 --> n

soit au total: n(n + 3)/2 < n2 pour n > 3.

Algorithmes de tris

- **Les algorithmes de tri**, sont utilisés principalement pour les tableaux (et les listes), ils peuvent aller du plus simple au plus compliquer.

- **Donnée** : Un tableau de n éléments $(a_0, a_1, \ldots, a_{n-1})$
- **Résultat** : Une permutation $(a'_0, a'_1, \ldots, a'_{n-1})$ du tableau d'entrée telle

$$a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1} \text{ ou } a'_0 \geq a'_1 \geq \ldots \geq a'_{n-1}$$

- L'opération de base pour les algorithmes de tri est la **comparaison** : A[i] : A[j],
- Une autre opération est **l'échange** : on peut échanger A[i] et A[j], ce que l'on note par A[i] $\leftrightarrow$ A[j]. Le résultat est alors le suivant :

$k \leftarrow A[i]$

$A[i] \leftarrow A[j]$

$A[j] \leftarrow k$

Algorithmes de tris

Le tri à bulle

- C'est un des algorithmes le plus connu moins efficace mais correcte.
- Le **principe** consiste à **balayer** tout le tableau, en **comparant** les éléments **adjacents** et en les **échangeant** s'ils ne sont pas dans le bon ordre. Ce processus est **répété** à chaque fin de tableau jusqu'à aucun échange ne sera effectué.
- Ce tri va nécessiter un **grand** nombre de déplacements d'éléments, il est donc inutilisable dans les cas où ces déplacements sont **coûteux** en temps.
- Il peut être intéressant quand le tableau initial est **pré-trié** (classement alphabétique)

Algorithmes de tris

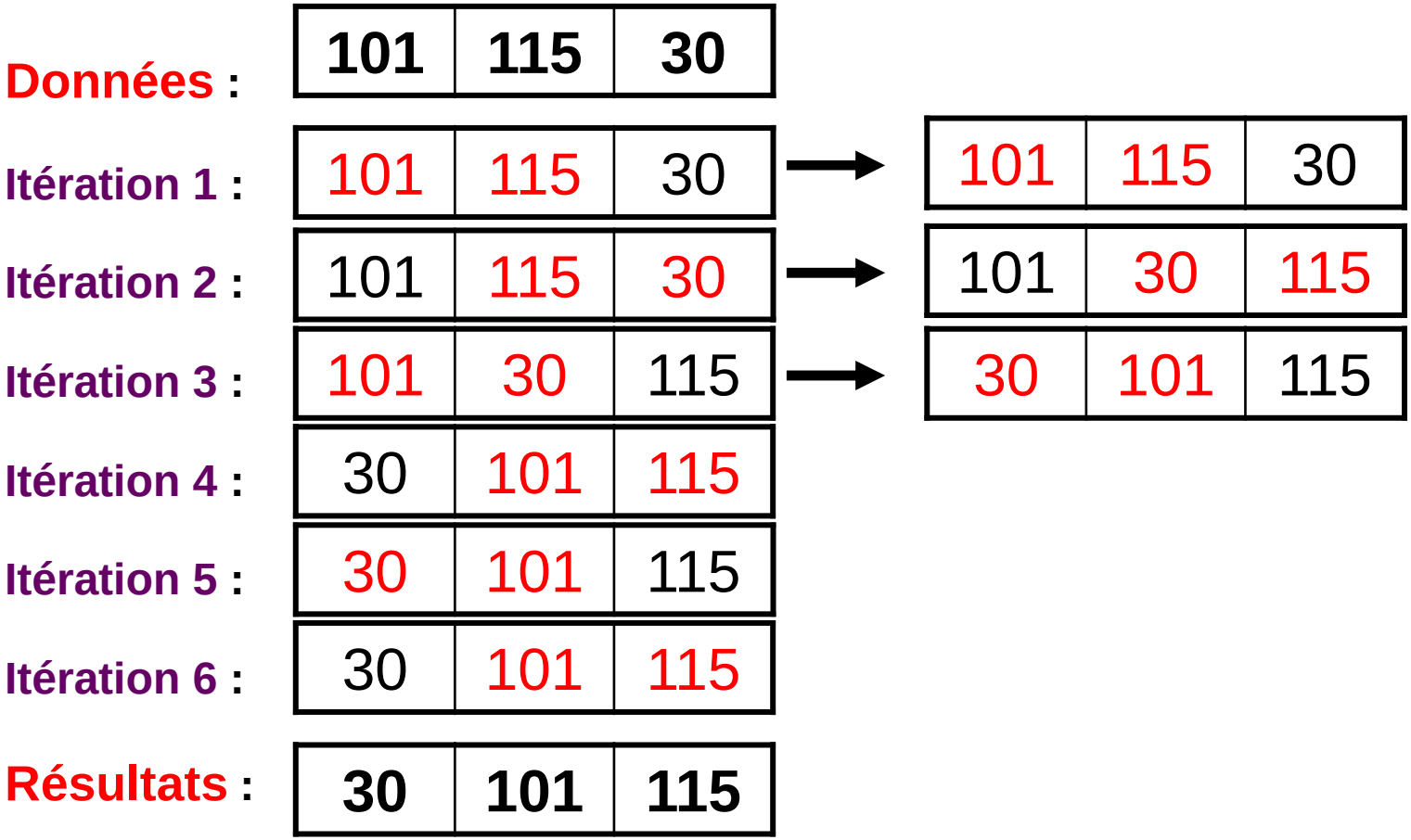
Le tri à bulle

```
Algorithme : TRI-BULLE
Données : A: tableau [0..n-1]  : entier
Variable : i, x : entier
           Echange  : booléen
début
        Répéter
                Echange ← Faux
                pour i de 1 à n-2 faire
                        Si A[i-1]>A[i]  alors
                                Echange ← Vrai
                                x ← A[j]
                                A[j] ← A[i]
                                A[i] ← x
                        finSi
        jusqu'à  non Echange
fin
```

Algorithmes de tris

Le tri à bulle

Exemple



Algorithmes de tris

Le tri par insertion

- Prendre les éléments du tableau **l'un après l'autre** dans l'ordre initial, et les placer **correctement** dans les éléments précédents déjà triés (classement des cartes de jeux après une donne).

- A **l'itération k**, le $k^{ème}$ élément est inséré d'une manière **séquentielle** parmi les premiers (k − 1) éléments qui sont déjà triés, cet l'opération est répété **n** fois (n taille du tableau)

- Le tri par insertion peut être intéressant pour des tableaux ayant déjà été **triés**, mais où l'on doit rajouter quelques nouveaux éléments.

- **Complexité** : le nombre de comparaison à l'itération k est **au plus k**, le nombre total de comparaison est au plus **(n(n+1)/2)-1** ∈ **Θ(n2)**

Algorithmes de tris

Le tri par insertion

```
Algorithme : TRI-INSERTION
Données : A: tableau [0..n-1]  : entier
Variable : i, x, k : entier
début
        pour i de 1 à n-1 faire
                k←i-1
                x ← A[i]
                Tant que A[k]>x  faire
                        A[k+1] ← A[k]
                        k ← k-1
                finTq
                A[k+1] ← x
        finPour
fin
```

Algorithmes de tris

Le tri par insertion

Exemple :

Données :	**101**	**115**	**30**	**63**	**47**	**20**
Itération 1 :	101	115	30	63	47	20
Itération 2 :	101	115	30	63	47	20
Itération 3 :	30	101	115	63	47	20
Itération 4 :	30	63	101	115	47	20
Itération 5 :	30	47	63	101	115	20
Itération 6 :	20	30	47	63	101	115
Résultats :	**20**	**30**	**47**	**63**	**101**	**115**

Algorithmes de tris

Le tri par sélection

- Le but est désormais de déplacer chaque élément à sa position **définitive**.

- **Rechercher** de l'élément le plus petit (ou plus grand) et **l'échanger** avec l'élément de la première position de la liste de recherche, l'opération est répétée **n-1 fois**

- Après le **$k^{ème}$ placement**, les k plus petits éléments du tableau sont à leur place définitive, il faut recommencer avec la liste des éléments restants du tableau

- **Complexité** : le nombre de comparaison à l'itération k est **n-k**, le nombre total de comparaison est **n(n-1)/2** $\in \Theta(n^2)$

Algorithmes de tris

Le tri par sélection

```
Algorithme : TRI-INSERTION
Données : A: tableau [0..n-1]  : entier
Variable : i, x, k,j : entier
Début
        i←0
        Tant que i<n-1 faire      { ième placement }
                j←i
                pour k de i+1 à n-1 faire
                        Si A[k]<A[j] faire
                                j ← k
                finPour
                x ← A[j]
                A[j] ← A[i]
                A[i] ← x
        finTq
fin
```

Algorithmes de tris

Le tri par sélection

Exemple :

Données :	**101**	**115**	**30**	**63**	**47**	**20**
Itération 1 :	20	115	30	63	47	101
Itération 2 :	20	30	115	63	47	101
Itération 3 :	20	30	47	63	115	101
Itération 4 :	20	30	47	63	115	101
Itération 5 :	20	30	47	63	101	115
Résultats :	**20**	**30**	**47**	**63**	**101**	**115**

Algorithmes de tris

Le tri rapide (Quick Sort)

- Ce tri est **récursif**. On cherche à trier une partie du tableau, délimitée par les indices gauche et droite.
- Un **pivot** est choisi aléatoirement parmi les valeurs du tableau
- Recherche de la **position définitive** de ce pivot de telle sorte que tous les éléments avant le pivot soient plus petit que lui et que tous ceux placés après lui soient supérieurs (mais sans chercher à les classer)
- Rappelle récursif du tri de la partie avant le pivot et de la partie après le pivot jusqu'à que les parties ne contiennent qu'un **seul élément**
- Complexité en pire cas : $\Theta(n^2)$ et en meilleur cas : $\Theta(n \times \log(n))$

Algorithmes de tris

Le tri rapide (Quick Sort)

```
Algorithme : TRI-RAPIDE
Données : A: tableau [0..n-1] , gch, drt  : entier
Variable : i, j, x, pivot : entier
début
        i← gch          j ← drt          pivot ← A[(i+j)/2]
        répéter
                tant que t[i]<pivot faire
                        i ← i+1
                tant que t[j]> pivot faire
                        j ← j-1
                si i<= j alors
                        echanger(A[i],A[j])
                        i ← i+1
                        j ← j-1
                finsi
        jusqu'à ce que i>j
```

Algorithmes de tris

Le tri rapide (Quick Sort)

Algorithme : **TRI-RAPIDE**
Données : A: **tableau** [0..n-1] , gch, drt : **entier**
Variable : i, j, x, pivot : **entier**
Début (suite)

```
        si gch < j alors
                Tri-Rapide(A,gch,j)
        finsi
        si i < drt alors
                Tri-Rapide(A,i,droit)
        finsi
```

fin

Algorithmes de tris

Le tri rapide

Exemple :

Données :	**101(i)**	**115**	**30(p)**	**63**	**47**	**20 (j)**
Itération 1 :	20	115(i)	30	63	47(j)	101
Itération 2 :	20	47	30(i)	63(j)	115	101
Itération 3 :	20	47(j)	30	63(i)	115	101

Puis relancer le processus sur

	20	47		63	115	101
Résultats :	**20**	**30**	**47**	**63**	**101**	**115**

Algorithmes de tris

Le tri Fusion (mergesort)

- Application de la méthode **diviser pour régner** au problème du tri d'un tableau
- Le **principe** est de diviser une table de longueur n en deux tables de longueur **n/2**, de trier **récursivement** ces deux tables, puis de **fusionner** les tables triées.
- Complexité en pire cas : $\Theta(n^2)$ et en meilleur cas : $\Theta(n \times \log(n))$

Algorithmes de tris

Le tri Fusion (mergesort)

Première phase : Découpe :

1. **Découper** le tableau en 2 sous-tableaux égaux (à 1 case près)
2. **Découper** chaque sous-tableau en 2 sous-tableaux égaux
3. Ainsi de suite, jusqu'à obtenir des sous-tableaux de **taille 1**

Deuxième phase : Tri/fusion :

1. **Fusionner** les sous-tableaux **2 à 2** de façon à obtenir des sous-tableaux de **taille 2** triés
2. Fusionner les sous-tableaux **2 à 2** de façon à obtenir des sous-tableaux de **taille 4** triés
3. Ainsi de suite, jusqu'à obtenir le tableau **entier**

Algorithmes de tris

Le tri Fusion (mergesort)

Algorithme : **FUSION**

Données : A: **tableau** [0..n-1] , a, b,c : **entier**

Variable : i,j,k : **entier**

A1 : **tableau[0…b-a**] , A2 :**tableau** [0…c-b-1] :**entier**

début

pour i **de** 0 **à** b-a **faire**

A1[i]← A[a+i]

pour j **de** 0 **à** c-b-1 **faire**

A2[j] ← A[b+1+j]

i ← 0

j ← 0

k ← a

Algorithmes de tris

```
Algorithme : FUSION(suite)
        tant que (k <= c) faire
                si (i >= b-a) alors
                        A[k] ← A2[j]
                         j ← j+1
                sinon
                        si (j >= c-b-1) alors
                                A[k] ← A1[i]          i ← i+1
                        sinon
                                si (A1[i] <= A2[j]) alors
                                        A[k] ← A1[i]        i ← i+1
                                sinon
                                        A[k] ← A2[j]        j ← j+1
                                finsi
                        finsi
                finsi
                k++
        fintantque
fin
```

Algorithmes de tris

Le tri Fusion (mergesort)

Algorithme : **TRI-FUSION**

Données : A: **tableau** [0..n-1] , a, b : **entier**

début

si (b>a) **alors**

triFusion(A,a,(a+b)/2)

triFusion(A,((a+b)/2)+1,b)

fusion(A,a,(a+b)/2,b)

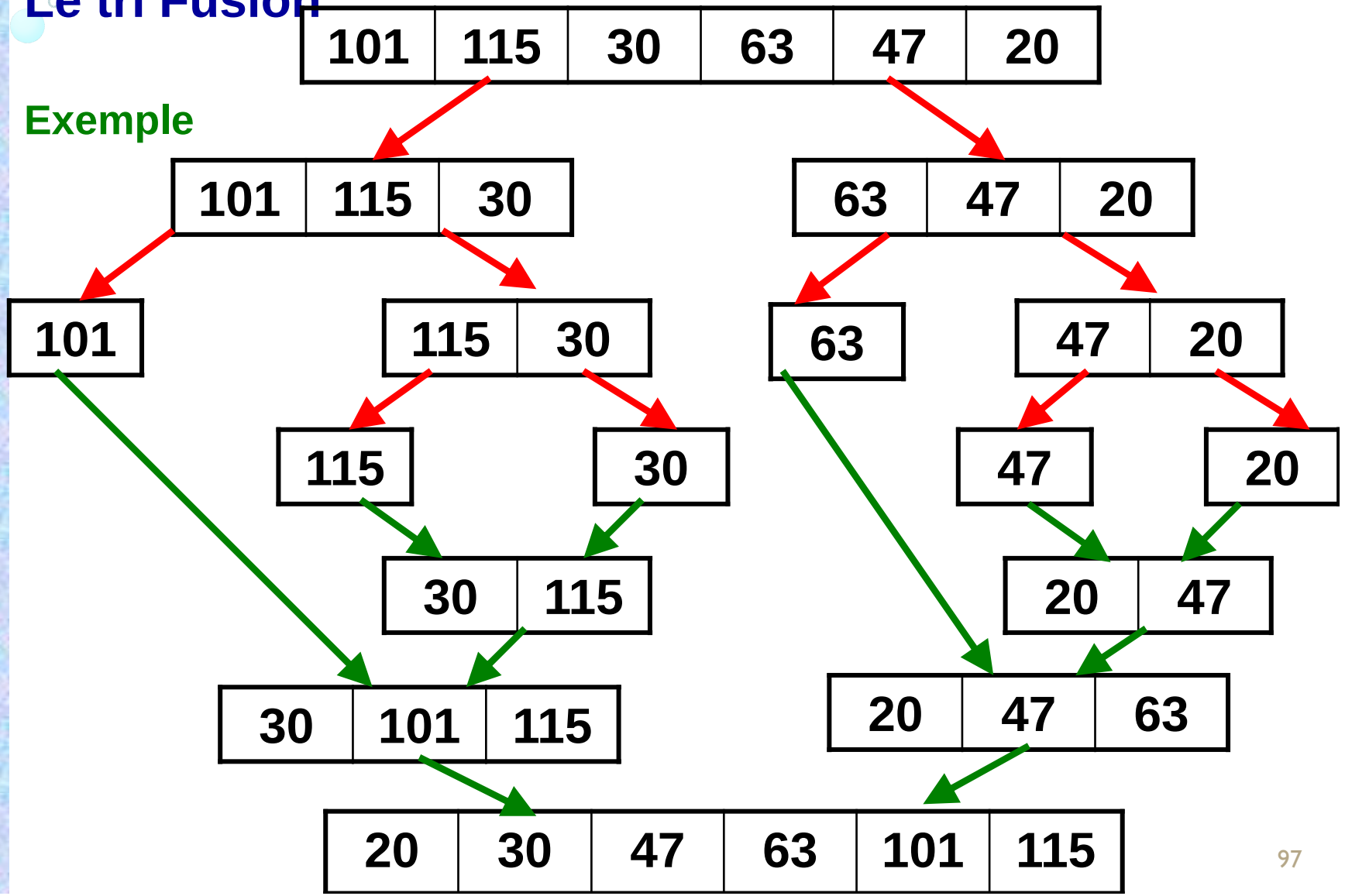
finsi

fin

Algorithmes de tris

Le tri Fusion

Exemple



Algorithmes de tris

La recherche séquentielle

- A partir d'un tableau trié, on **parcours ce tableau élément par élément** jusqu'à trouver le bon élément.

La recherche dichotomique

- La recherche dichotomique recherche un élément dans un tableau trié et retourne **l'indice d'une occurrence** de cet élément.
- Le **principe** consiste à compare l'élément cherché à celui qui se trouve au **milieu** du tableau. Si l'élément cherché est plus petit, alors continuer la recherche dans la **première moitié** du tableau, sinon dans la **seconde moitié**.
- Recommencer ce processus sur la moitié et s'arrêter lorsque l'on a **trouvé l'élément**, ou lorsque **l'intervalle** de recherche est **nul**.