



Algorithmique Avancée

Pr. : Essaid SABIR

Contact : e.sabir@ensem.ac.ma

Cycle : Ingénieur

Niveau : Ing1

Année : 2017/2018



- Généralités
- Programmation informatique
- Algorithmique Classique
- Récursivité
- Algorithmes de Tri
- Complexité et Optimalité
- Structures de Données
- Arbres de Recherche
- Graphes



Algorithmique

Définition d'un algorithme

- Le terme **algorithme** vient du nom du mathématicien arabe **Al Khawarizmi** (820 après J.C.)
- Un algorithme est une description complète et détaillée des actions à effectuer et de leur séquencement pour arriver à un résultat donné
 - **Intérêt:** séparation analyse/codage (pas de préoccupation de syntaxe)
 - **Qualités:** **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), ...
- **L'algorithmique** désigne aussi la discipline qui étudie les algorithmes et leurs applications en Informatique
- Une bonne connaissance de l'algorithmique permet d'écrire des algorithmes exacts et efficaces



Historiquement, deux façons pour représenter un algorithme:

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

Algorithmique

Notion de variable



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

Algorithmique



Choix du nom d'une variable

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

- Un nom doit commencer par une lettre alphabétique
exemple valide: A1 **exemple invalide: 1A**
 - doit être constitué uniquement de lettres, de chiffres et du soulignement _
(Eviter les caractères de ponctuation et les espaces)

valides: GI2018, GI_2017

invalides: GI 2016, GI-2017, GI;2018

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

Algorithmique

Choix du nom d'une variable



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

Exemples: TotalVentes2017, Prix_TTC, Prix_HT

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



Algorithmique

Type d'une variable

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

- **Type numérique (entier ou réel)**

- **Byte** (codé sur 1 octet): de 0 à 255
- **Entier court** (codé sur 2 octets) : -32 768 à 32 767
- **Entier long** (codé sur 4 ou 8 octets)
- **Réel simple précision** (codé sur 4 octets)
- **Réel double précision** (codé sur 8 octets)

- **Type logique ou booléen** : deux valeurs VRAI ou FAUX

- **Type caractère** : lettres majuscules, minuscules, chiffres, symboles, ...

Exemples: 'A', 'a', '1', '?', ...

- **Type chaîne de caractère** : toute suite de caractères,

Exemples: " Nom, Prénom", "code postale: 1000", ...



- **Rappel:** toute variable utilisée dans un programme doit être déclarée au préalable
- En pseudo-code, on adopte la forme suivante pour la déclaration de variables

Variables liste d'identificateurs : type

- **Exemple :**

Variables i, j,k : entier

x, y : réel

OK : booléen

ch1, ch2 : chaîne de caractères

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

Algorithmique

L'affectation



- **l'affectation** consiste à attribuer une valeur à une variable (ça consiste en fait à remplir où à modifier le contenu d'une zone mémoire)
- En pseudo-code, l'affectation se note avec le signe \leftarrow

Var \leftarrow x: attribue la valeur de e à la variable Var

- x 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

- **Exemples valides :**

i \leftarrow 1 j \leftarrow i k \leftarrow i+j

x \leftarrow 10.3 OK \leftarrow FAUX ch1 \leftarrow »GC2»

ch2 \leftarrow ch1 x \leftarrow 4 x \leftarrow j

Algorithmique

L'affectation



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

Algorithmique

L'affectation



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

Variables A, B, C: Entier

Début

A \leftarrow 3

B \leftarrow 7

A \leftarrow B

B \leftarrow A+5

C \leftarrow A + B

C \leftarrow B - A

Fin



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

Variables A, B : Entier

Début

A \leftarrow 1

B \leftarrow 2

A \leftarrow B

B \leftarrow A

Fin

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



Exercice

Ecrire un algorithme permettant d'échanger les valeurs de deux variables A et B.

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

Algorithmique

Les opérateurs



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

- \wedge : (élévation à la puissance)
- $*$, $/$ (multiplication, division)
- $\%$ (modulo)
- $+$, $-$ (addition, soustraction)

Exemple : $2 + 3 * 7$ vaut 23

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

Exemple : $(2 + 3) * 7$ vaut 35

Algorithmique

Lecture/Ecriture d'une valeur



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

Algorithmique

Lecture/Ecriture d'une valeur



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

Algorithmique

Lecture/Ecriture d'une valeur



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

Algorithme Calcul_double

variables A, B : entier

Début

écrire("entrer le nombre ")

lire(A)

B \leftarrow 2*A

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

Fin



Algorithmique

Lecture/Ecriture d'une valeur

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

Algorithme AffichageNomComplet

variables Nom, Prenom, Nom_Complet : chaîne de caractères

Début

```
écrire("entrez votre nom »)
lire(Nom)
écrire("entrez votre prénom »)
lire(Prenom)
Nom_Complet ← Nom & Prenom
écrire("Votre nom complet est : ", Nom_Complet)
```

Fin

Algorithmique



Instruction conditionnelle

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



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

Si condition alors

instruction ou suite d'instructions1

Finsi

Algorithmique

Instruction conditionnelle



Exemple. Valeur absolue d'une nombre réel

Algorithme AffichageValeurAbsolue (version1)

Variable x : réel

Début

Ecrire (" Entrez un réel : ")

Lire (x)

Si (x < 0) **alors**

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

Sinon

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

Finsi

Fin

Algorithmique

Instruction conditionnelle



Exemple. Valeur absolue d'une nombre réel

Algorithme AffichageValeurAbsolue (version2)

Variable x,y : réel

Début

Ecrire (" Entrez un réel : ")

Lire (x)

y \leftarrow x

Si ($x < 0$) alors

y \leftarrow -x

Finsi

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

Fin

Algorithmique

Instruction conditionnelle



Exercice.

Ecrire un algorithme qui demande un nombre entier à l'utilisateur, puis qui teste et affiche s'il est divisible par 5

Algorithme Divsible_par5

Variable n : entier

Début

Ecrire " Entrez un entier : "

Lire (n)

Si (n%5=0) **alors**

Ecrire (n," est divisible par 5")

Sinon

Ecrire (n," n'est pas divisible par 5")

Finsi

Fin



- 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**

Algorithmique

Instruction conditionnelle: Condition composée



Tables de vérité

| C1 | C2 | C1 ET C2 |
|------|------|-------------|
| VRAI | VRAI | VRAI |
| VRAI | FAUX | FAUX |
| FAUX | VRAI | FAUX |
| FAUX | FAUX | FAUX |

| C1 | C2 | C1 OU C2 |
|------|------|-------------|
| VRAI | VRAI | VRAI |
| VRAI | FAUX | VRAI |
| FAUX | VRAI | VRAI |
| FAUX | FAUX | FAUX |

| C1 | C2 | C1 XOR C2 |
|------|------|-------------|
| VRAI | VRAI | FAUX |
| VRAI | FAUX | VRAI |
| FAUX | VRAI | VRAI |
| FAUX | FAUX | FAUX |

| C1 | NON C1 |
|------|-------------|
| VRAI | FAUX |
| FAUX | VRAI |

Algorithmique

Instruction conditionnelle: 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

Algorithmique

Instruction conditionnelle: Tests imbriqués



Exemple (version 1) : Signe d'un nombre

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

Algorithmique

Instruction conditionnelle: Tests imbriqués



Exemple (version 2)

Variable n : entier

Début

Ecrire ("entrez un nombre : ")
Lire (n)

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

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

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

Finsi

Fin

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

Conseil : utiliser les tests imbriqués pour limiter le nombre de tests et placer d'abord les conditions les plus probables



Exercice.

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

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

Algorithmique

Instruction conditionnelle: Tests imbriqués



Exercice (solution).

Variables copies : entier
 prix : réel

Début

Ecrire ("Nombre de photocopies : ")

Lire (copies)

Si (copies < 10) **Alors**
 prix ← copies*0.5

Sinon Si (copies) < 20
 prix ← copies*0.4

Sinon

 prix ← copies*0.3

Finsi

Finsi

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

Fin



Les boucles servent à répéter l'exécution d'un groupe d'instructions un certain nombre de fois

On distingue trois types de boucles en langages de programmation :

- La **boucle tant que** : on y répète des instructions tant qu'une certaine condition est réalisée
- La **boucle jusqu'à** : on y répète des instructions jusqu'à ce qu'une certaine condition soit réalisée
- La **boucle pour** ou avec compteur : on y répète des instructions en faisant évoluer un compteur (variable particulière) entre une valeur initiale et une valeur finale

Algorithmique

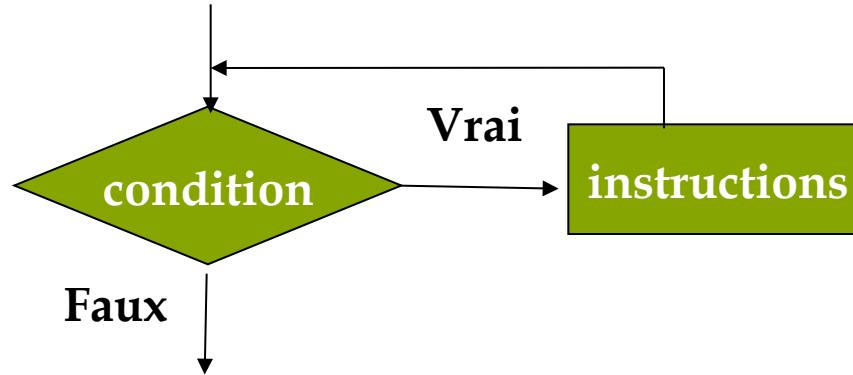
Instructions répétitives : Boucle Tant que



TantQue (condition)

instructions

FinTantQue



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



- Le nombre d'itérations dans une boucle TantQue n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment

⇒ **Attention aux boucles infinies**

- Exemple de boucle infinie :

$i \leftarrow 2$

TantQue ($i > 0$)

$i \leftarrow i+1$ (attention aux erreurs de frappe : + au lieu de -)

FinTantQue

Algorithmique

Instructions répétitives : Boucle Tant que

Exemple 1.

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



Variable C : caractère

Debut

Ecrire (" Entrez une lettre majuscule ")

Lire (C)

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

Ecrire ("Saisie erronée. Recommencez")

Lire (C)

FinTantQue

Ecrire ("Saisie valable")

Fin

Algorithmique

Instructions répétitives : Boucle Tant que



Exemple 2.

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

version 1

Variables som, i : entier

Debut

i \leftarrow 0

som \leftarrow 0

TantQue (som \leq 100)

 i \leftarrow i+1

 som \leftarrow som+i

FinTantQue

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

Fin



Algorithmique

Instructions répétitives : Boucle Tant que

Exemple 3.

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

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

Variables som, i : entier

Debut

 som \leftarrow 0

 i \leftarrow 1

TantQue (som $<=$ 100)

 som \leftarrow som + i

 i \leftarrow i+1

FinTantQue

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

Fin

Algorithmique

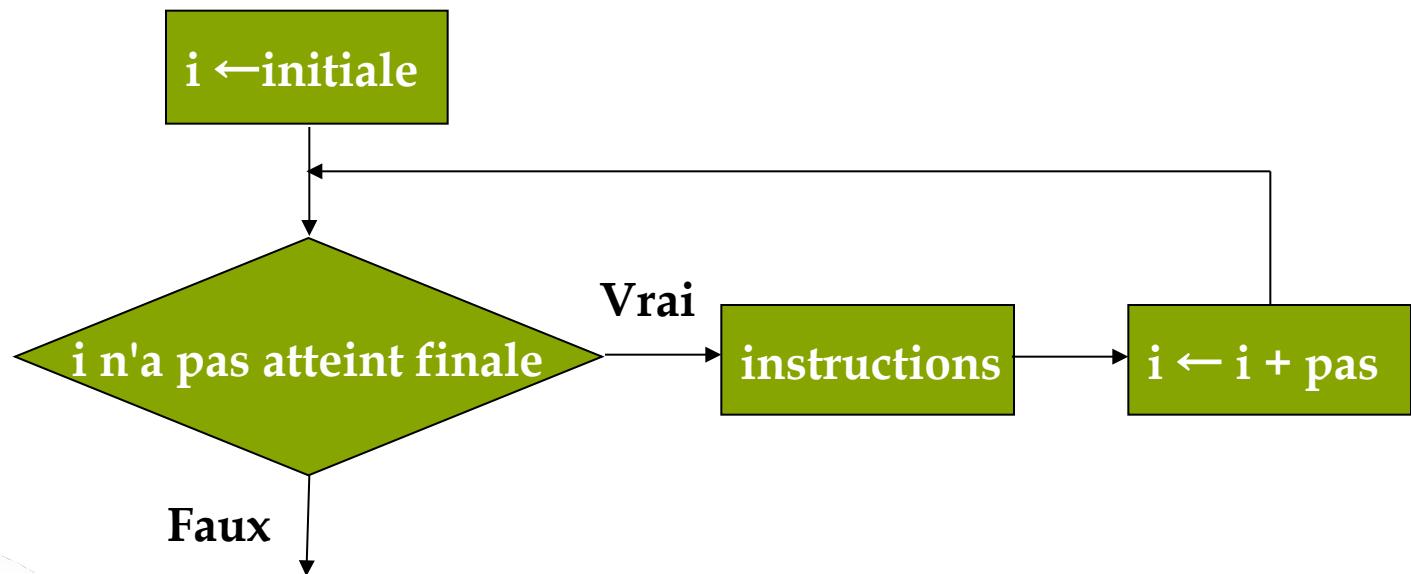
Instructions répétitives : Boucle Pour



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

Instructions

FinPour





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

Algorithmique

Instructions répétitives : Boucle Pour



Exemple 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 la valeur de x ")

Lire (x)

Ecrire (" Entrez la valeur de n ")

Lire (n)

$puiss \leftarrow 1$

Pour i allant de 1 à n

$puiss \leftarrow puiss * x$

FinPour

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

Fin



Algorithmique

Instructions répétitives : Boucle Pour

Exemple 2.

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

Variables x , puiss : réel

n , i : entier

Debut

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

Lire (x , n)

puiss \leftarrow 1

Pour i allant de n à 1 par pas -1

puiss \leftarrow puiss \ast x

FinPour

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

Fin



Remarques.

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

Exemple :

Pour i allant de 1 à 5

i ← i -1

écrire(" i = ", i)

Fin pour

Algorithmique

Instructions répétitives : Boucle Répéter ... jusqu'à ...

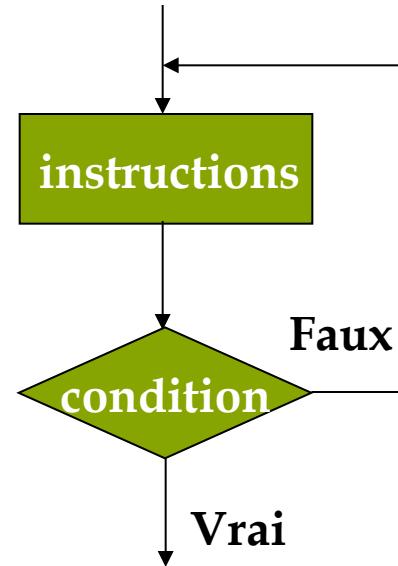


Répéter

instructions

Jusqu'à

condition



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



Algorithmique

Instructions répétitives : Boucle Répéter ... jusqu'à ...

Exemple.

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

Variables som, i : entier

Debut

 som \leftarrow 0

 i \leftarrow 0

Répéter

 i \leftarrow i+1

 som \leftarrow som+i

Jusqu'à (som > 100)

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

Fin



Discussion.

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

S'il n'est pas possible de connaître le nombre d'itérations avant l'exécution de la boucle, on fera appel à l'une des *boucles TantQue ou répéter jusqu'à*

Pour le choix entre *TantQue* et *jusqu'à* :

- Si on doit tester la condition de contrôle avant de commencer les instructions de la boucle, on utilisera *TantQue*
- Si la valeur de la condition de contrôle dépend d'une première exécution des instructions de la boucle, on utilisera *répéter jusqu'à*

Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**

Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :

- permettent de "factoriser" les **programmes**, c-à-d de mettre en commun les parties qui se répètent
- permettent une **structuration** et une **meilleure lisibilité** du code
- **facilitent la maintenance** du code (il suffit de modifier une seule fois)
- ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes



Algorithmique

Fonctions & Procédures : Fonction

Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat à partir des valeurs des paramètres**

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

Fonction nom_fonction (paramètres et leurs types) : type_fonction

Instructions constituant le corps de la fonction

retourne ...

FinFonction

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

Algorithmique

Fonctions & Procédures : Fonction

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

Fonction SommeCarre (x : réel, y: réel) : réel

```
variable z : réel  
z ← x^2+y^2  
retourne (z)
```

FinFonction

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

Fonction Pair (n : entier) : booléen

```
retourne (n%2=0)
```

FinFonction



Algorithmique

Fonctions & Procédures : Fonction

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 exempleAppelFonction

variables z : réel, b : booléen

Début

```
b ← Pair(3)
z ← 5 * SommeCarre(7,2) + 1
écrire("SommeCarre(3,5)= ", SommeCarre(3,5))
```

Fin

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



Algorithmique

Fonctions & Procédures : Procédure

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

Procédure nom_procédure (paramètres et leurs types)

Instructions constituant le corps de la procédure

FinProcédure

- Remarque : une procédure peut ne pas avoir de paramètres



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

Procédure exemple_proc (...)

...

FinProcédure

Algorithme exepmleAppelProcédure

Début

 exemple_proc (...)

...

Fin

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

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

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

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

Algorithmique

Fonctions & Procédures : Passage de paramètres



Exemple.

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

$x \leftarrow x+1$

$y \leftarrow y+1$

FinProcédure

Algorithme Test_incremente1

variables n, m : entier

Début

$n \leftarrow 3$

$m \leftarrow 3$

incremente1(n, m)

écrire (" n= ", n, " et m= ", m)

résultat :
n=3 et m=4

Fin

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

Algorithmique

Fonctions & Procédures : Passage de paramètres par adresse



Exemple.

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

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

 som \leftarrow x+y

 prod \leftarrow 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 \leftarrow x

 x \leftarrow y

 y \leftarrow z

FinProcédure

Algorithmique

Fonctions & Procédures : Variable locale et variable globale



On peut manipuler 2 types de variables dans un module (procédure ou fonction) : des **variables locales** et des **variables globales**. Elles se distinguent par ce qu'on appelle leur **portée** (leur "champ de définition", leur "durée de vie")

Une **variable locale** n'est connue qu'à l'intérieur du module où elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution

Une **variable globale** est connue par l'ensemble des modules et le programme principal. Elle est définie durant toute l'application et peut être utilisée et modifiée par les différents modules du programme

Algorithmique

Fonctions & Procédures : Variable locale et variable globale



- 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



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

Exemple : Calcul du factorielle

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

Algorithmique

Récursivité



Exercice.

Ecrivez une fonction récursive (puis itérative) qui calcule le terme n de la suite de Fibonacci définie par : $U(0)=U(1)=1$

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

Fonction Fibo (n : entier) : entier

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

retourne (1)

Sinon

retourne (Fibo(n-1)+Fibo(n-2))

Finsi

FinFonction



Exercice.

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

Fonction Fibo (n : entier) : entier

Variables i, AvantDernier, Dernier, Actuel: entier

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

Finsi

AvantDernier \leftarrow 1, Dernier \leftarrow 1

Pour i allant de 2 à n

Actuel \leftarrow Dernier + AvantDernier

AvantDernier \leftarrow Dernier

Dernier \leftarrow Actuel

FinPour

retourne (Actuel)

FinFonction

Algorithmique

Récursivité



Exercice.

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

Procédure binaire (n : entier)

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

 binaire (n/2)

 écrire (n mod 2)

Finsi

FinProcédure



Algorithmique

Tableaux

Supposons qu'on veut conserver les notes d'une classe de 30 étudiants pour extraire quelques informations. Par exemple : calcul du nombre d'étudiants ayant une note supérieure à 10

Le seul moyen dont nous disposons actuellement consiste à déclarer 30 variables, par exemple **N1, ..., N30**. Après 30 instructions lire, on doit écrire 30 instructions Si pour faire le calcul

```
nbre ← 0  
Si (N1 >10) alors nbre ←nbre+1 FinSi  
....  
Si (N30>10) alors nbre ←nbre+1 FinSi
```

c'est lourd à écrire

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



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



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

Algorithmique

Tableaux



Exercice.

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

Variables $i, nbre : entier$

tableau notes[30] : réel

Début

$nbre \leftarrow 0$

Pour i allant de 0 à 29

Si (notes[i] >10) alors

$nbre \leftarrow nbre+1$

FinSi

FinPour

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

Fin

Algorithmique

Tableaux : Saisie / Affichage de valeurs



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

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

Pour i allant de 0 à n-1
 écrire ("Saisie de l'élément ", i + 1)

 lire (T[i])

FinPour

Fin Procédure

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

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

FinPour

Fin Procédure



- Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par **deux indices**. Ceci est utile par exemple pour représenter des matrices
- En pseudo code, un tableau à deux dimensions se **déclare** ainsi :

variable **tableau** identificateur[dimension1] [dimension2] : type

- **Exemple :** Une matrice A de 3 lignes et 4 colonnes dont les éléments sont réels
variable **tableau** A[3][4] : réel
- **A[i][j]** permet d'accéder à l'élément de la matrice qui se trouve à l'intersection de la ligne i et de la colonne j

Algorithmique

Tableaux : Tableaux à deux dimensions



Procédure qui permet de saisir les éléments d'une matrice

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

Début

variables *i,j* : entier

Pour *i* allant de 0 à *n*-1
 écrire ("saisie de la ligne ", *i* + 1)

Pour *j* allant de 0 à *m*-1

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

 lire (*A*[*i*][*j*])

FinPour

FinPour

Fin Procédure

Algorithmique

Tableaux : Tableaux à deux dimensions



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

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

variables i,j : entier

Pour i allant de 0 à n-1
Pour j allant de 0 à m-1

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

FinPour

FinPour

Fin Procédure

Algorithmique

Tableaux : Tableaux à deux dimensions



Exercice.

Procédure qui calcule la somme de deux matrices

Procédure SommeMatrices(n, m : entier par valeur,

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

Début

variables i,j : entier

Pour i allant de 0 à n-1

Pour j allant de 0 à m-1

$C[i][j] \leftarrow A[i][j] + B[i][j]$

FinPour

FinPour

Fin Procédure



- **Recherche d'un élément dans un tableau**

- Recherche séquentielle
- Recherche dichotomique

- **Tri d'un tableau**

- Tri par sélection (minimum successif)
- Tri à bulles
- Tri rapide

Algorithmique

Tableaux : Recherche d'une valeur



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

Variables i : entier, Trouvé : booléen

Début

$i \leftarrow 0$, Trouvé \leftarrow Faux

TantQue ($i < N$) ET (Trouvé=Faux)

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

 Trouvé \leftarrow Vrai

 Sinon

$i \leftarrow i+1$

 FinSi

FinTantQue

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

 écrire ("x appartient au tableau")

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

 Sinon

 FinSi

Fin

Algorithmique

Tableaux : Recherche d'une valeur



Une fonction Recherche qui retourne un booléen pour indiquer si une valeur x appartient à un tableau T de dimension N .

x , N et T sont des paramètres de la fonction

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

Variable i : entier

Pour i allant de 0 à $N-1$

Si ($T[i]=x$) **alors**
retourne (Vrai)

FinSi

FinPour

retourne (Faux)

FinFonction



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



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

| | | | |
|-------|-----|----|---------|
| temps | 1ms | 1s | 16mn40s |
|-------|-----|----|---------|



- Dans le cas où le tableau est ordonné, on peut améliorer l'efficacité de la recherche en utilisant la méthode de recherche dichotomique

- **Principe :** diviser par 2 le nombre d'éléments dans lesquels on cherche la valeur x à chaque étape de la recherche. Pour cela on compare x avec $T[\text{milieu}]$:
 - Si $x < T[\text{milieu}]$, il suffit de chercher x dans la 1ère moitié du tableau entre ($T[0]$ et $T[\text{milieu}-1]$)

 - Si $x > T[\text{milieu}]$, il suffit de chercher x dans la 2ème moitié du tableau entre ($T[\text{milieu}+1]$ et $T[N-1]$)

Algorithmique

Tableaux : Efficacité d'un algorithme



gauche \leftarrow 0 , droite \leftarrow N-1, Trouvé \leftarrow Faux

TantQue (gauche \leq droite) ET (Trouvé = Faux)
milieu \leftarrow (gauche + droite)/2

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

 Trouvé \leftarrow Vrai

SinonSi (T[milieu] < x) **alors**
 gauche \leftarrow milieu + 1

Sinon droite \leftarrow milieu - 1

FinSi

FinSi

FinTantQue

Si Trouvé **alors** écrire ("x est trouvé dans le tableau")

Sinon écrire ("x n'a pas été trouvé dans le tableau")

FinSi

Algorithmique

Tableaux : Efficacité d'un algorithme



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



- Le tri consiste à ordonner les éléments du tableau dans l'ordre croissant ou décroissant
- Il existe plusieurs algorithmes connus pour trier les éléments d'un tableau :
 - Le tri par sélection
 - Le tri par insertion
 - Le tri rapide
 - ...
- Nous verrons dans la suite l'algorithme de tri par sélection et l'algorithme de tri rapide. Le tri sera effectué dans l'ordre croissant



- **Principe :** Le tri par minimum successif est un tri par sélection,
Pour une place donnée, on sélectionne l'élément qui doit y être positionné
- De ce fait, si on parcourt la tableau de gauche à droite, on positionne à chaque fois le plus petit élément qui se trouve dans le sous tableau droit
- Ou plus généralement : Pour trier le sous-tableau $T[i\dots N]$ il suffit de positionner au rang i le plus petit élément de ce sous-tableau et de trier le sous-tableau $T[i+1\dots N]$

Algorithmique

Tableaux : Tri par sélection



Exemple.

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

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

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

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

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

- **Étape 3:**

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

Algorithmique

Tableaux : Tri par sélection



Supposons que le tableau est noté T et sa taille N

Pour i allant de 0 à N-2

 indice_ppe \leftarrow i

Pour j allant de i + 1 à N-1

 Si $T[j] < T[\text{indice_ppe}]$ alors

 indice_ppe \leftarrow j

 Finsi

 FinPour

 temp $\leftarrow T[\text{indice_ppe}]$
 $T[\text{indice_ppe}] \leftarrow T[i]$
 $T[i] \leftarrow \text{temp}$

FinPour

Algorithmique

Tableaux : Tri par sélection (complexité)



- Quel que soit l'ordre du tableau initial, le nombre de tests et d'échanges reste le même
- On effectue $N-1$ tests pour trouver le premier élément du tableau trié, $N-2$ tests pour le deuxième, et ainsi de suite.
 - ➔ Soit : $t(N) = (N-1)+(N-2)+\dots+1 = N(N-1)/2$
 - ➔ On effectue en plus $e(N) = (N-1)$ échanges (pire des cas).
- La complexité du tri par sélection est en **$O(N^2)$** à la fois dans le meilleur des cas, en moyenne et dans le pire des cas
- Pour un ordinateur qui effectue 10^6 tests par seconde on a :

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



Algorithmique

Tableaux : Tri à bulles

- **Principe :** Sélectionner le minimum du tableau en parcourant le tableau de la fin au début et en échangeant tout couple d'éléments consécutifs non ordonnés.
- Exemple, pour trier $\langle 101, 115, 30, 63, 47, 20 \rangle$, on va avoir les boucles suivantes :
- $i=1 \quad \langle 101, 115, 30, 63, 47, \textcolor{red}{20} \rangle$
 $\langle 101, 115, 30, \textcolor{red}{20}, 63, 47 \rangle$
 $\langle 101, 115, \textcolor{red}{20}, 30, 63, 47 \rangle$
 $\langle 101, \textcolor{red}{20}, 115, 30, 63, 47 \rangle$
- $i=2 \langle \textcolor{red}{20}, 101, 115, 30, 63, 47 \rangle$
- $i=3 \langle 20, 30, 101, 115, 47, 63 \rangle$
- $i=4 \langle 20, 30, 47, 101, 115, 63 \rangle$
- $i=4 \langle 20, 30, 47, 63, 101, 115 \rangle$ (**fin du tri**)

Algorithmique

Tableaux : Tri à bulles (algorithme)



Supposons que le tableau est noté T et sa taille N

Pour i allant de 0 à N-1 faire

Pour k allant de N-1 à i+1 faire

Si T[k]<T[k-1] alors

 permuter(T[k],T[k-1])

FinSi

FinPour

FinPour

Algorithmique

Tableaux : Tri à bulles (complexité)



- Nombre de tests(moyenne et pire des cas) :

$$t(N)=N+t(N-1)$$

$$\rightarrow t(N)= N(N+1)/2$$

→ Complexité en $O(N^2)$.

- Nombre d'échanges (pire des cas):

$$e(n) = N - 1 + N - 2 + \dots + 1$$

→ Complexité en $O(N^2)$

- Nombre d'échanges (en moyenne): $O(N^2)$ (calcul plus complexe)

- **Verdict :** Complexité en $O(N^2)$.

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

Algorithmique

Tableaux : Tri rapide



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

variable q: entier

Si p < r **alors**

 Partition(T,p,r,q)

 TriRapide(T,p,q-1)

 TriRapide(T,q+1,r)

FinSi

Fin Procédure

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

Algorithmique



Tableaux : Tri rapide

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

Variables i, j : entier pivot: réel

`pivot← T[p], i←p+1, j ← r`

TantQue (i<=j)

TantQue ($i \leq r$ et $T[i] \leq \text{pivot}$) $i \leftarrow i+1$

FinTantQue

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

FinTantQue

Si $i < j$ alors

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

FinSi

FinTantQue

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

$q \leftarrow j$

Fin Procédure

Algorithmique

Tableaux : Tri rapide (complexité)



- La complexité du tri rapide dans **le pire des cas** est en $O(N^2)$
- La complexité du tri rapide en **moyenne** est en $O(N \log N)$
- Le choix du pivot influence largement les performances du tri rapide
- Le pire des cas correspond au cas où le pivot est à chaque choix le plus petit élément du tableau (tableau déjà trié)
- différentes versions du tri rapide sont proposés dans la littérature pour rendre le pire des cas le plus improbable possible, ce qui rend cette méthode la plus rapide en moyenne parmi toutes celles utilisées

Crédits & Références



- Mouad Ben Mamoun & Moulay Driss Rahmani, "Cours d'Informatique"

- <https://www.wikipedia.org>