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;

Langage C : opérateurs

- **Instructions de base**
  - opérateurs de base
    - +, -, *, / → opérateurs arithmétique de base
    - % → reste d'une division entière
    - == → test d'égalité
    - != test de différence
    - <, >, <=, >= →test de comparaison
    - ! → négation
    - || → **ou** logique pour évaluer une expression
    - && → **et** logique pour évaluer une expression

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; **}**

Langage C : syntaxe des tests

Écriture en pseudo code

cas où v **vaut**
    v1 : action1
    v2 : action2
    . . .
    vn : actionn
  autre : action autre
Fincas

Traduction en C

```
switch(v){
  case v1 : action1; break;
  case v2 : action2; break;
  …
  case vn: actionn ;break;
  default : action autre; break;
}
```

Langage C : syntaxe des boucles

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

Fonctions et procédures

Les procédures et les fonctions

- Par exemple, pour 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 de naturels avec des notes saisies par l'utilisateur
  - Afficher un tableau de naturels
  - Trier un tableau de naturel en ordre croissant
  - Trouver le plus petit naturel d'un tableau
  - Trouver le plus grand naturel d'un tableau
  - Calculer la moyenne d'un tableau de naturels

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

Fonctions et 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. Elles ont plusieurs avantages :
  - permettent d'éviter de réécrire un même traitement plusieurs fois. En effet, on fait appelle 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

Fonctions

- Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique : elle **retourne un résultat au programme appelant**
- 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 constituant le corps de la fonction
    retourne //la valeur à retourner
FinFonction
```

- Le nom_fonction est un identificateur
- type_fonction est le type du résultat retourné
- L'instruction retourne sert à retourner la valeur du résultat

Caractéristiques des fonctions

- 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,…)

Fonctions : exemples

- 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>y) 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

Utilisation des fonctions

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

```
variables 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

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

  Procédure nom_procédure (paramètres et leurs types)

  Variables //locales

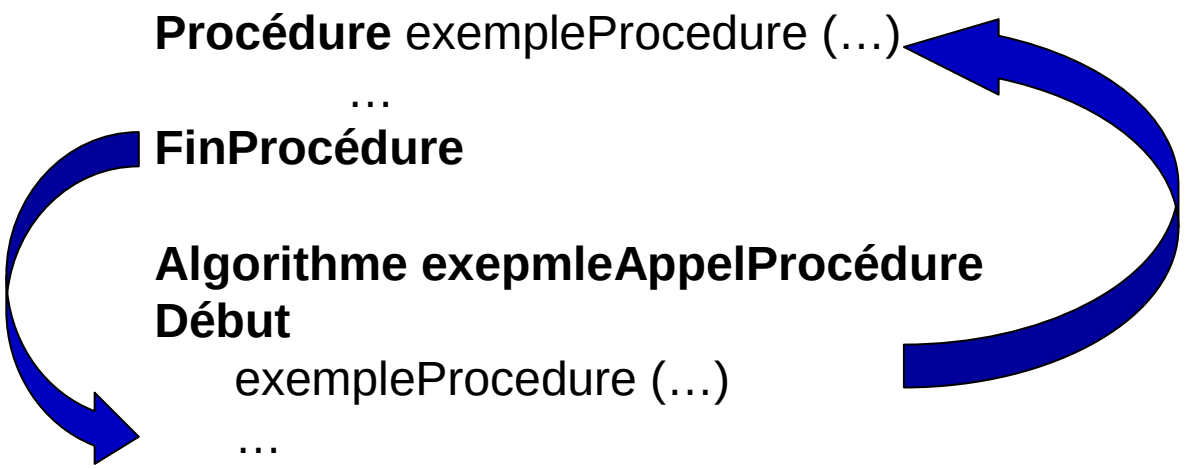
  Début

  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 :



- 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
- Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre

Transmission des paramètres

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

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 ou elle a été définie. Elle est créée à l'appel du module et détruite à la fin de son exécution

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

Variables locales et globales

- 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)
  { ensemble instructions
  }
  - 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;
}
```

Récursivité

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

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

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
        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
        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 )
        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 de 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, elle ne sont pas indispensable.

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.
  - 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

- 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**

Tableaux

- Un **tableau** est un ensemble d'éléments de même type désignés par un identificateur unique
- Une variable entière nommée **indice** permet d'indiquer la position d'un élément donné au sein du tableau et de déterminer sa valeur
- La **déclaration** d'un tableau s'effectue en précisant le **type** de ses éléments et sa **dimension** (le nombre de ses éléments)
  - En pseudo code :

    variable **tableau** identificateur**[dimension] : type**
  - Exemple :

    variable **tableau** notes**[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]

  Pour accéder au ième élément, on écrit tab[i-1] (avec 0<i<=10)

Tableaux : remarques

- 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 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
- 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 modifiée le contenu du 7ème élément du tableau (43 au lieu de 49)

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 allant de 0 à N-1
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 =200 : 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 allant de 0 à n-1 faire
                ecrire("entrer une note :")
                lire(Notes[i])
        finpour
                /* effectuer la moyenne des notes */
        somme ← 0
        pour i allant de 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 allant de 0 à n-1
                écrire ("Saisie de l'élément ", i + 1)
                lire (T[i] )
        FinPour

/* affichage  */
        Pour i allant de 0 à n-1
                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 de 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 allant de 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 } ;**

- Déclaration d'un tableau de plusieurs dimensions
  - type nom_tableau[dim1][dim2]…[dimn];
  - Exemple: char buffer[20][80];

Tableaux à deux dimensions

- Les langages de programmation permettent de déclarer des tableaux dans lesquels les valeurs sont repérées par **deux indices**. Ceci est utile par exemple pour représenter des matrices
- En pseudo code, un tableau à deux dimensions se **déclare** ainsi :
  variable **tableau** identificateur**[dimension1] [dimension2] : type**
  - Exemple : une matrice A de 3 linges 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
- 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, n, m : entier
  tableau A[N][N] : réel
  Début
  ecrire("entrer le nombre de lignes et le nombre de colonnes :")
  lire(n, m)
  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

Exemples : affichage d'une matrice

- Affichages des éléments d'une matrice :
- **Constante** N=100 : entier
  Variable i, j, n,m : 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(n, m)
        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
```

Initialisation de matrice

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

```
T1[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
T2[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
T3[4][4] = {{1,2,3}, {4,5,6}, {7,8,9}};
T4[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 allant de 0 à n-1
       Pour j allant de 0 à m-1
                  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 allant de 1 à n faire
                écrire("donner les éléments de la ",i," ligne:")
                pour j allant de 1 à n faire
                                        lire(A[i][j])
                finpour
        finpour
                                        /* lecture de la matrice B */
        pour i allant de 1 à n faire
                écrire("donner les éléments de la ",i," ligne:")
                pour j allant de 1 à n faire
                                        lire(B[i][j])
                finpour
        finpour
```

Exemples : produit de deux matrices (suite)

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

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 de nos programmes.
- Le temps d'exécution dépend de plusieurs facteurs :
  - Les données (trier 4 nombre ce n'est pas trier 1000)
  - Le code généré par le compilateur
  - La nature de la machine utilisée
  - La complexité de l'algorithme.
- Si T(n) dénote le temps d'exécution d'un programme sur un ensemble des données de taille n alors :

Complexité d'un algorithme

- $T(n)=c.n^2$ (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 ∃ c et n0 telles que ∀ n≥n0, T(n) ≤ c.f(n). L'algorithme ayant T(n) comme temps d'exécution a une complexité O(f(n))

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

- 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
- La recherche 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

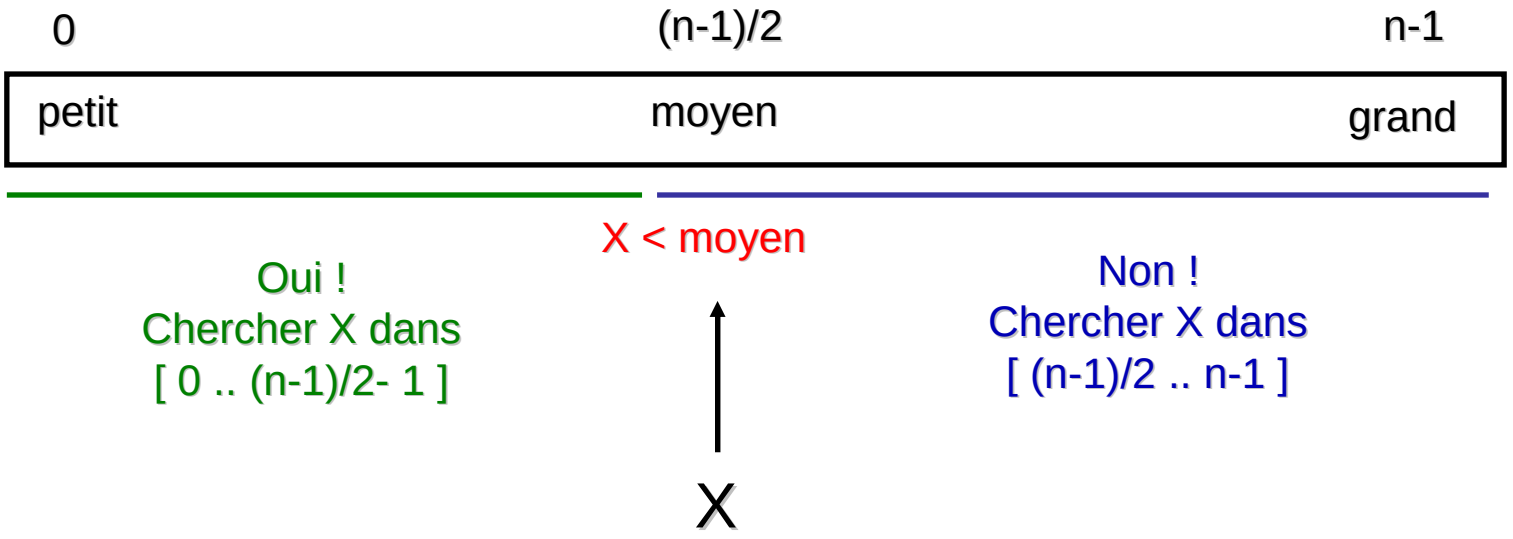
- Dans le pire des cas on doit parcourir tout le tableau. Ainsi, la complexité est de l'ordre de O(n).
- Si le te tableau est trié la recherche séquentielle peut s'arrêter dés qu'on rencontre un élément du tableau strictement supérieur à l'élément recherché.
- Si tous les éléments sont plus petits que 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
- **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 Trouvé)
        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]
  - milieu
  - [milieu+1, sup]

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

  Cas 3 : sup-milieu ≤ sup -(inf+sup)/2 ≤ (sup-inf)/2
- On passe dons 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.
- Le nombre d'éléments à la k ième itération est : $(½)^{k-1}n$ 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 repartir 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].

- **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:

2	5	9	8	6

  - **Étape 3:**

2	5	6	8	9

Tri par sélection-échange : algorithme

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

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

Fin à n-2 !

Chercher l'indice du plus petit à partir de i.

Echange, même si i = indice_ppe.

Tri par sélection-échange : complexité

- On fait n-1 fois, pour i de 0 à n-2 :
- Un parcours de [i..n-1].
- Il y a donc un nombre de lectures qui vaut :

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

Tri en complexité quadratique.

Tri par insertion

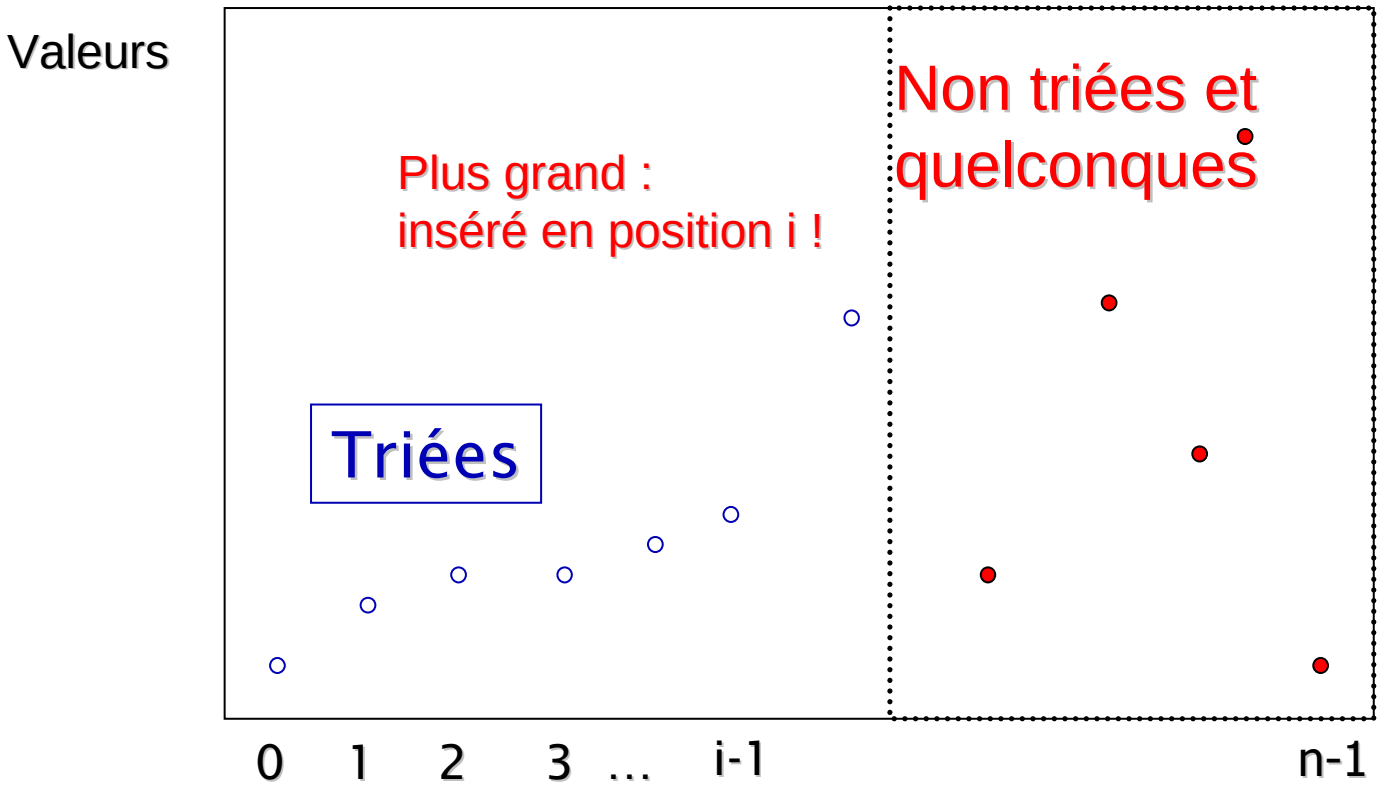
À la ième étape :

- Cette méthode de tri insère le ième élément T[i-1] à la bonne place parmi T[0], T[2]…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.

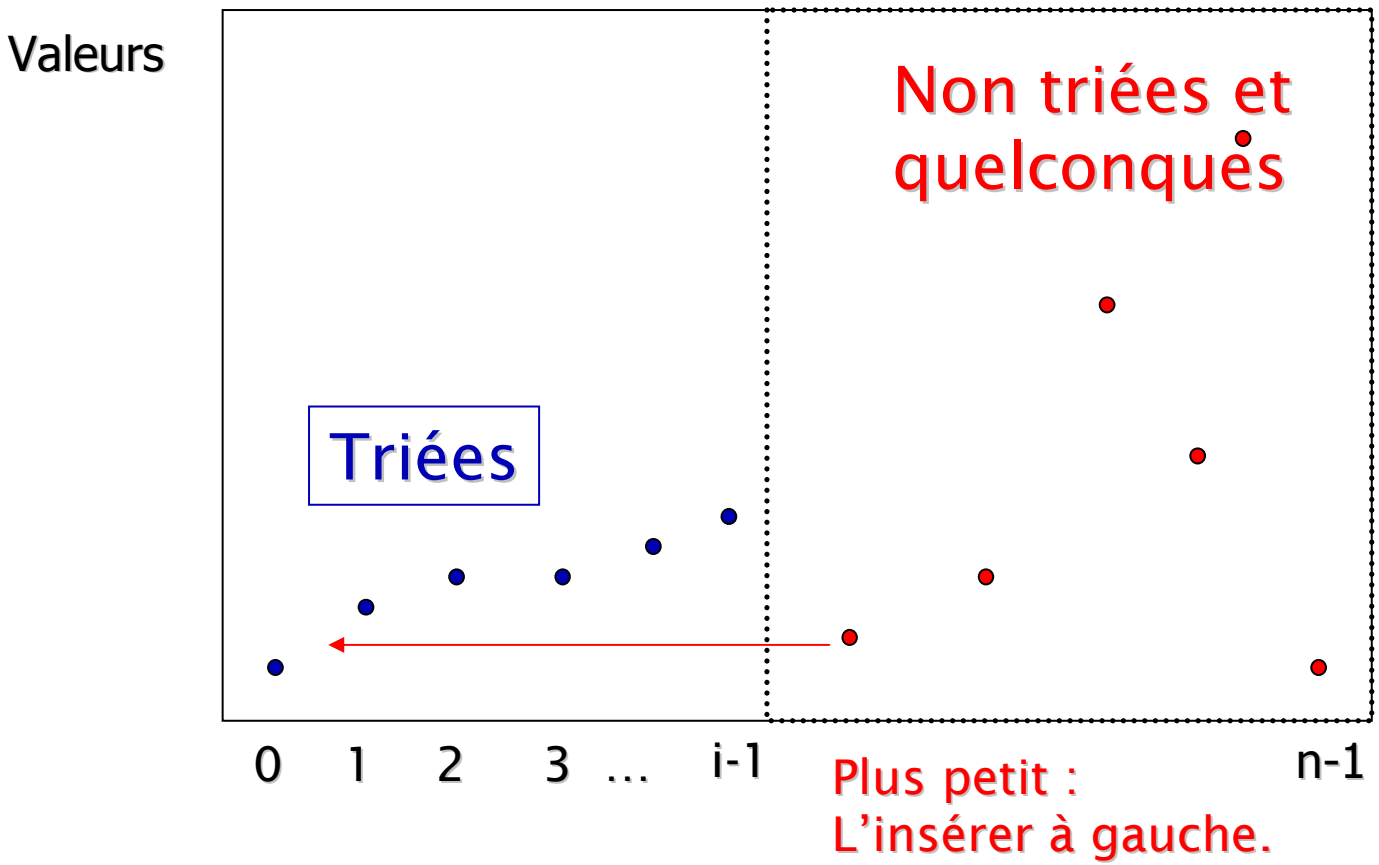
Pour insérer l'élément T[i-1] :

- 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

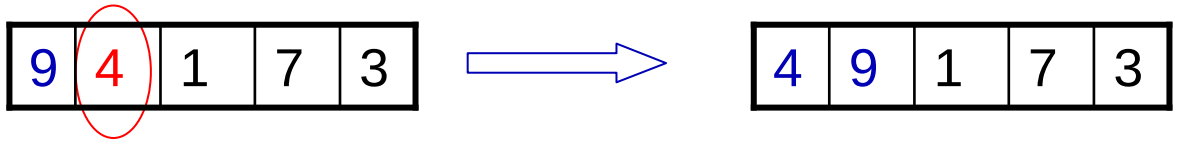


Tri par insertion

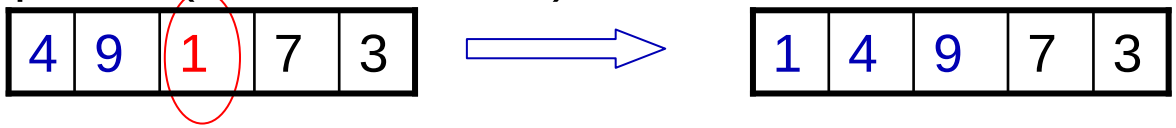


Tri par insertion : exemple

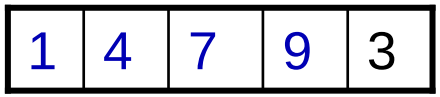
- **Étape 1:** on commence à partir du 2 ième élément du tableau (élément 4). On cherche à l'insérer à la bonne position par rapport au sous tableau déjà trié (formé de l'élément 9) :



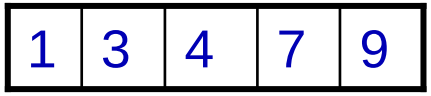
- **Étape 2:** on considère l'élément suivant (1) et on cherche à l'insérer dans une bonne position par rapport au sous tableau trié jusqu'à ici (formé de 4 et 9):



- **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
        decaler ←vraie; j ←i
        Tantque ((j >0) et (decaler))
                Si T[j] <T[j-1] alors  temp ← T[j]
                                       T[j] ← T[j-1]
                                       T[j-1] ← temp
                sinon decaler ←faux
                Finsi
                j←j-1;
        FinTantque
FinPour
```

le premier élément est forcément à sa place

On échange aussi longtemps que cela est possible

Tri par insertion : la complexité

- On fait n-1 fois, 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 considère un élément du tableau qu'on appelle pivot
  - **2)** on partitionne le tableau en 2 sous tableaux : les éléments inférieurs ou égaux au pivot et 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

Procédure Tri rapide

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

variable q: entier
Si p <r **alors**
Partition(T,p,r,q)
TriRapide(T,p,q-1)
TriRapide(T,q+1,r)
FinSi
Fin Procédure

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

Procédure de partition

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

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

Tri rapide : 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 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

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

Enregistrements

- Les langages de programmation offrent, en plus des types de base (entier, réel, booléen…), 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
- 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

Utilisation des enregistrements

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