

# Introduction

## Bases de l'algorithmique et algorithmique avancée

Sylvain Contassot-Vivier



- L'informatique met en jeu des ordinateurs qui fonctionnent selon des schémas pré-établis
- Pour résoudre un problème donné à l'aide d'un ordinateur, il faut lui indiquer la *suite d'actions* à exécuter dans son schéma de fonctionnement
- Cette suite d'actions est un *programme* qui est exprimé dans un langage de programmation plus ou moins évolué (code machine, assembleur, C, Caml, Prolog...)
- Pour écrire un programme (la suite d'actions), il faut donc d'abord savoir *comment faire* pour résoudre le problème



## Algorithme

- Un *algorithme* est l'expression de la résolution d'un problème de sorte que le résultat soit calculable par machine
- L'algorithme est exprimé dans un modèle théorique de machine universelle (Von Neumann) qui ne dépend pas de la machine réelle sur laquelle on va l'utiliser
- Il peut être écrit en langage naturel, mais pour être lisible par tous, on utilise un *langage algorithmique* plus restreint qui comporte tous les concepts de base de fonctionnement d'une machine
- On a finalement l'enchaînement suivant :  
Énoncé du problème → Algorithme → Programme  
(universel) (lié à une machine)

## Problème et énoncé

- Un problème a un *énoncé* qui donne des informations sur le *résultat* attendu
- Il peut être en langage naturel, imprécis, incomplet et ne donne généralement pas la façon d'obtenir le résultat
- Exemples :
  - Calculer le PGCD de 2 nombres
  - Calculer la surface d'une pièce
  - Ranger des mots par ordre alphabétique
  - Calculer  $x$  tel que  $x$  divise  $a$  et  $b$  et si  $y$  divise  $a$  et  $b$  alors  $x \geq y$
- Le dernier exemple décrit très précisément le résultat mais ne nous dit pas *comment le calculer*



# Résolution de problèmes

- Pour résoudre un problème, il faut donner *la suite d'actions élémentaires* à réaliser pour obtenir le résultat
- Les actions élémentaires dont on dispose sont définies par le langage algorithmique utilisé
- Exemple :



Parfois, il faut *décomposer* les actions trop complexes

# Un exemple

- *L'ordre* des opérations a son importance, mais dans certains cas plusieurs ordres sont possibles
- *Algorithme d'Euclide* : Calcule le PGCD de 2 entiers a et b

Ordonner les deux nombres tel que  $a \geq b$   
Calculer leur différence  $c = a - b$   
Recommencer en remplaçant a par c ( $a = c$ )  
jusqu'à ce que a devienne égal à b

# Exécution avec 174 et 72

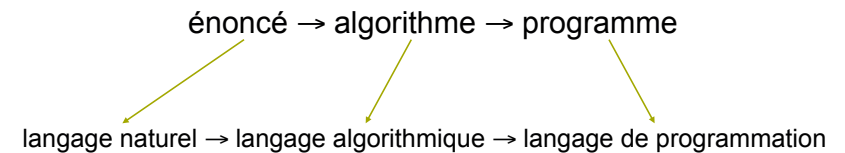
Étape 1 :  $a=174$  et  $b=72$   
Étape 2 :  $c=174-72=102$   
Étape 3 :  $a=c=102 \neq b=72$   
Étape 4 :  $a=102$  et  $b=72$   
Étape 5 :  $c=102-72=30$   
Étape 6 :  $a=c=30 \neq b=72$   
Étape 7 :  $a=72$  et  $b=30$   
Étape 8 :  $c=72-30=42$   
Étape 9 :  $a=42 \neq b=30$   
Étape 10 :  $a=42$  et  $b=30$   
Étape 11 :  $c=42-30=12$   
Étape 12 :  $a=12 \neq b=30$   
...

...

Étape 13 :  $a=30$  et  $b=12$   
Étape 14 :  $c=30-12=18$   
Étape 15 :  $a=18 \neq b=12$   
Étape 16 :  $a=18$  et  $b=12$   
Étape 17 :  $c=18-12=6$   
Étape 18 :  $a=6 \neq b=12$   
Étape 19 :  $a=12$  et  $b=6$   
Étape 20 :  $c=12-6=6$   
Étape 21 :  $a=6 = b=6$

Arrêt car  $a=b$   
le résultat est donc 6

# Langages



- Un langage est composé de deux éléments :
  - La *grammaire* qui fixe la syntaxe
  - La *sémantique* qui donne le sens
- Le langage répond donc à deux questions :
  - Comment écrire les choses ?
  - Que signifient les choses écrites ?

# Types de langages

- Langages machines : code binaire, liés au processeur
- Langages assembleurs : bas niveau, liés au processeur
- Langages évolués : haut niveau, indépendants de la machine
  - ⇒ compilateur ou interpréteur pour l'exécution
  - *Langages impératifs (procéduraux)* : suite des instructions à réaliser dans l'ordre d'exécution : C, Pascal, Fortran, Cobol
  - *Langages à objets* : modélisation par entités ayant des propriétés et des *interactions* possibles : C++, Java
  - *Langages déclaratifs* : règles de calcul et vérification de propriétés sans spécification d'ordre :
    - Fonctionnels : les règles sont exprimées par des fonctions : Caml
    - Logiques : les règles sont exprimées par des prédicats logiques : Prolog

## Description du résultat

- Spécifier précisément ce que doit produire l'algorithme et le lien entre le résultat et les données fournies au départ
- **Synopsis** :
  - Utilisation du mot clé **Résultat** : suivi d'un texte en langage naturel décrivant le résultat
- **Exemples** :
  - Problème** : trouver  $x$  tel que  $x$  divise  $a$  et  $b$  et si  $y$  divise  $a$  et  $b$  alors  $x \geq y$
  - Résultat** : affichage du PGCD de deux entiers  $a$  et  $b$
  
  - Problème** : calculer la surface d'un champ
  - Résultat** : affichage de la surface d'un champ

# Étapes de la conception

- La conception de l'algorithme est la phase la plus difficile
- On privilégie une méthodologie *hiérarchique* en *décomposant* l'algorithme en parties
- Chaque partie est elle-même décomposée jusqu'à obtenir des instructions élémentaires (*raffinements successifs*)
- Les différents éléments d'un algorithme :
  - Description du résultat : ce que doit produire l'algo
  - Idée de l'algorithme : les grandes étapes de traitement/calcul
  - Lexique des variables : liste des valeurs manipulées
  - Algorithme : le détail du traitement/calcul
- D'autres éléments peuvent s'ajouter (on les verra plus loin...)

## Idée de l'algorithme

- Description informelle en langage naturel des grandes étapes de l'algorithme (1ère décomposition)
- **Synopsis** :
  - Mot clé **Idée** : suivi de l'énumération des étapes
- Exemple du calcul de la surface d'un champ rectangulaire  
Idée :
  - Acquérir la longueur et la largeur du champ
  - Calculer la surface
  - Afficher la surface
- Ces 3 étapes sont *déduites* du résultat demandé
- Chaque étape de l'idée peut elle-même être décomposée si elle est trop complexe, et ainsi de suite (cf la maison)

# Lexique des variables

- Liste des valeurs manipulées par l'algorithme
- **Synopsis** :
  - Mot clé **Lexique des variables** : suivi de la liste détaillée des variables
- Les variables permettent de nommer et mémoriser les valeurs manipulées par l'algorithme
- Elles sont définies par :
  - **Un nom** : référence de la variable. Exemple : numéroProduit  
cf formalisation des noms de variables
  - **Un type** : nature de la valeur (entier, réel, caractère,...)  
⇒ cohérence des calculs/traitements

# Nommage des variables

- La convention de nommage que nous utiliserons est la suivante :
  - Toujours donner un nom **représentatif** de la sémantique de la variable
  - Commencer par une lettre minuscule
  - Lorsque le nom contient plusieurs mots, on met une majuscule au début de chaque mot interne pour faciliter la lecture :
    - Exemples :
      - Une variable représentant un nombre d'étudiants pourra être nommée **nombreEtudiants** ou encore **nbEtu**
      - Une variable représentant la surface d'un polygone sera nommée **surfacePolygone**, **surfacePoly** ou encore **surfPoly**
  - Les abréviations de mots (surf pour surface, nb pour nombre,...) sont autorisées tant qu'elles n'engendrent pas d'ambiguïté de sens
  - Exceptionnellement, une seule lettre sera admise dans certains cas tels que les compteurs de boucle ou les variables mathématiques

# Lexique des variables

- Les informations données pour chaque variable sont :
  - **Nom** : en suivant les conventions spécifiées
  - **Type** : indiqué entre parenthèses
  - **Description** : texte bref en langage naturel donnant la signification de la variable dans l'algorithme
  - **Rôle** : on distingue trois rôles possibles :
    - **DONNÉE** : la valeur de la variable est **donnée** à l'algorithme par le biais d'une lecture depuis un média quelconque
    - **RÉSULTAT** : la valeur de la variable est fournie en **résultat** de l'algorithme, elle est généralement calculée par celui-ci
    - **INTERMÉDIAIRE** : la valeur de la variable est calculée par l'algorithme et utilisée dans des calculs sans être fournie en résultat
    - Dans certains cas, des variables peuvent avoir les deux rôles Don/Rés

# Lexique des variables

- Exemple du calcul de la surface d'un champ rectangulaire :  
Lexique des variables :

NOM	TYPE	DESCRIPTION	RÔLE
surface	(réel)	Surface du champ	RÉSULTAT
longueur	(réel)	Longueur du champ	DONNÉE
largeur	(réel)	Largeur du champ	DONNÉE

- La largeur et la longueur du champ sont des données du problème
- La surface est le résultat rendu par l'algorithme
- Il n'y a pas de variable intermédiaire dans cet algorithme

# Algorithmme

- Détail en langage algorithmique des traitements/calculs effectués par l'algorithme
- **Synopsis** :
  - Mot clé **Algorithme** : suivi de la suite des actions élémentaires
- Exemple : la surface d'un champ  
Algorithme :

```
longueur ← lire
largeur ← lire
surface ← longueur * largeur
écrire surface
```

# Explications

- L'affectation : indiquée par ←
  - Affecte la valeur à droite de la flèche dans la variable placée à gauche
  - `truc ← machin` peut se lire : « variable *truc* reçoit la valeur *machin* »
  - Attention à la correspondance des types !
    - On ne peut affecter à une variable qu'une valeur de *même type*
  - Permet de vérifier la cohérence de ce que l'on écrit
- Exemple :  
`maVariable ← 15`
  - `maVariable` contient 15 après l'instruction
  - La valeur 15 sera prise en lieu et place de `maVariable` dans la suite de l'algorithme jusqu'à sa prochaine modification

# Opérateurs divers

- **lire** : signifie que la valeur est donnée par l'utilisateur
- **écrire** : affiche du texte et/ou des valeurs à l'écran
- Les opérateurs arithmétiques sont \* / + - et % (modulo)  
⚠ le type du résultat dépend du type des opérands :
  - Le résultat a toujours le type le plus complexe des opérands
  - Exemples :
    - entier *op* entier ⇒ entier
    - entier *op* réel ⇒ réel
    - réel *op* entier ⇒ réel
    - réel *op* réel ⇒ réel
    - entier / entier ⇒ entier (division entière !!)

# Opérateurs divers (suite)

- Les opérateurs booléens : et, ou, non dans {vrai,faux}
  - A et B : vrai si A **et** B sont vrais
  - A ou B : vrai si A est vrai **ou** B est vrai
  - non A : inverse logique de A (noté ¬)
- Opérateurs de comparaison : ≥ ≤ < > = ≠
- Autres fonctions mathématiques : on suit le C/C++

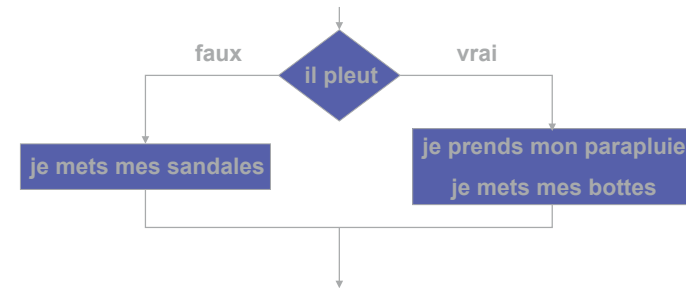
$e^x$	<code>exp(x)</code>
$x^y$	<code>pow(x,y)</code>
$\sqrt{x}$	<code>sqrt(x)</code>
$\lfloor x \rfloor$	<code>floor(x)</code>
$\lceil x \rceil$	<code>ceil(x)</code>
$  x  $	<code>abs(x)</code> ou <code>fabs(x)</code>

# Lexique des constantes

- Valeurs utilisées mais non modifiées par l'algorithme
- **Synopsis** :
  - Mot clé **Lexique des constantes** : suivi de la liste des constantes
- Placé **avant** le lexique des variables
- Elles sont définies par :
  - **Un nom** : référence de la constante. Exemple : TAUX\_TVA (tout en majuscules avec \_ pour séparer les mots)
  - **Un type** : nature de la valeur (entier, réel, caractère,...)
  - **Une valeur** : la valeur de la constante, connue **avant** l'exécution de l'algorithme et non modifiée par celui-ci
  - **Une description** : un texte indiquant ce que représente la constante

# Les conditionnelles

- Possibilité de **choisir** une séquence d'instructions selon une condition donnée
- Exemple :  
"s'il pleut, je prends mon parapluie et je mets mes bottes sinon je mets mes sandales"



# Conditionnelles

- **Synopsis** :

<pre>si &lt;condition&gt; alors   &lt;instruction&gt;   &lt;instruction&gt;   ... sinon   &lt;instruction&gt;   &lt;instruction&gt;   ... fsi</pre>	<pre>si &lt;condition&gt; alors   &lt;instruction&gt;   &lt;instruction&gt;   ... fsi</pre>
---	---
- La **condition** est une expression booléenne {vrai,faux}
- Si la condition est **vraie**, on exécute la branche **alors**
- Si la condition est **fausse**, on exécute la branche **sinon**

# Imbrication de conditionnelles

- Toute instruction algorithmique peut être placée dans une conditionnelle, donc également une conditionnelle !
- Cela permet de multiplier les choix possibles d'exécution
- Exemple :

```
si c1 alors
  ... /* c1 vraie */
sinon
  si c2 alors /* c1 fausse */
    ... /* c2 vraie */
  sinon
    ... /* c2 fausse */
  fsi
fsi
```
- L'ordre des imbrications est généralement important

# Enchaînement des conditions

- On utilise les opérateurs booléens pour lier plusieurs conditions :
  - val<0 *ou* val>20, cpt<10 *et* val>15
- Deux possibilités d'évaluation des conditions :
  - Totale** : *toutes* les conditions sont évaluées et les résultats sont combinés par les opérateurs booléens pour donner le résultat final
  - Minimale** : les conditions sont évaluées de gauche à droite tant que le résultat final n'est pas déterminé  
Dès qu'une condition détermine le résultat final, **arrêt de l'évaluation !**
  - Exemples : val>20 **non testé** si val<0, val>15 **non testé** si cpt≥10
- Cela a une incidence sur la façon d'écrire certaines structures de contrôle basées sur des test
- Dans la suite, nous considérerons l'**évaluation minimale** lorsque le mode d'évaluation ne sera pas précisé

# Conditionnelles de type cas

- Lorsque l'on veut comparer *une seule* variable à une **énumération** de valeurs *connues* à l'avance, on peut utiliser la structure de contrôle **selon que**

- Synopsis** :

```
selon que <variable> est
<val1> : <instruction>
...
<val2> : <instruction>
...
<valN> : <instruction>
...
défaut : <instruction>
...
fselon
```

les *val* sont des *constantes* toutes différentes mais du même type que *variable*

le cas *défaut* est optionnel et est choisi lorsqu'aucune des valeurs énumérées ne correspond à la variable

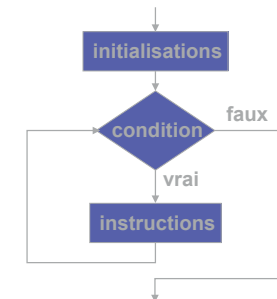
# Contrôles itératifs (boucles)

- Possibilité de **répéter** une suite d'instructions selon une condition donnée
- Exemple : Euclide ⇒ répéter ... jusqu'à ce que a=b
- On dispose de 3 structures de contrôles différentes :
  - Le **tant que** :
    - Répète des instructions tant que la condition de la boucle est *vraie*
    - Les instructions de la boucle peuvent *ne pas* être exécutées
  - Le **pour** :
    - Répète des instructions un nombre *connu* de fois
    - La condition porte uniquement sur le nombre d'itérations à effectuer
  - Le **répéter** :
    - Comme le **tant que** mais on effectue *au moins une fois* les instructions de la boucle

# Structure tant que

- Synopsis** :

```
<initialisations>
tant que <condition>
<instruction>
<instruction>
...
ftant
```



- Les *initialisations* spécifient les valeurs initiales des variables intervenant dans la *condition*
- Il faut *au moins une* instruction dans la boucle susceptible de modifier la valeur de la condition
- Les *instructions* de la boucle peuvent *ne pas* être exécutées



# Exemple : les factorielles

Algorithme :

```
n ← lire
fact ← 1
pour i de 1 à n
    fact ← fact * i
    écrire fact
fpour
```

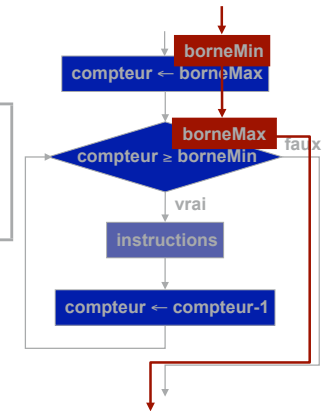
- Utilisation de la valeur de *i* dans la boucle
- Initialisation du ou des élément(s) calculé(s) dans la boucle, ici la variable *fact*
- Si on place l’affichage en dehors de la boucle (après le fpour) on affiche uniquement la dernière valeur calculée

# Sens du pour

- Par défaut, une boucle pour va dans le sens *croissant*
- On peut inverser le sens
- **Synopsis :**

```
pour <compteur> de <borneMax> à <borneMin> en descendant
    <instruction>
    <instruction>
    ...
fpour
```

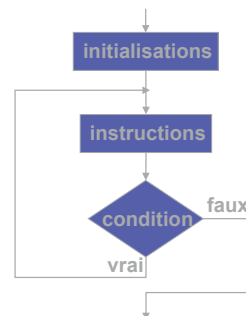
⚠ On n’entre pas dans la boucle si ses bornes sont dans l’ordre inverse de son sens !!



# Structure répéter

- Similaire au *tant que* mais exécutée *au moins une fois*
- **Synopsis :**

```
<initialisations>
répéter
    <instruction>
    <instruction>
    ...
tant que <condition>
```

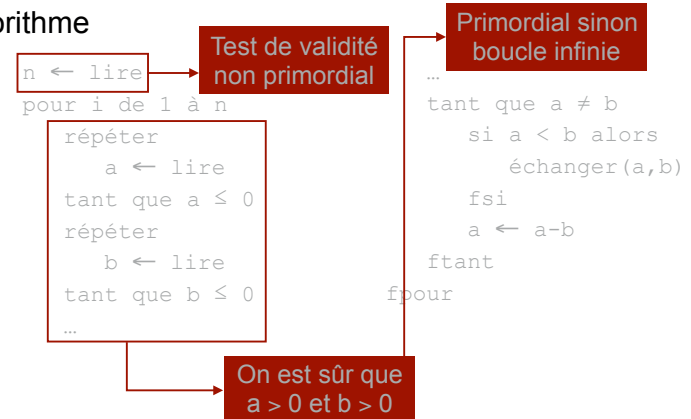


- Pratique pour la vérification des lectures de données :
  - répéter la lecture tant que celle-ci n’est pas *valide*

# Boucles imbriquées

- On peut placer n’importe quel type de boucle dans une autre
- Exemple :

Algorithme



# Arrêt sur la fin des données

- Permet de lire des données tant que l'utilisateur en fournit
  - Sans en connaître le nombre a priori
- Le booléen **\_échecLecture\_** est Vrai si la dernière opération de lecture a échoué (interruption utilisateur ou fin de données)
  - Lorsqu'une lecture échoue, la variable destination n'est pas modifiée
- Exemple : Algorithme

```
n ← lire
tant que ¬_échecLecture_
  fact ← 1
  pour i de 1 à n
    fact ← fact * i
  fpour
  écrire fact
  n ← lire
ftant
```

# Les fonctions

- Morceaux d'algorithmes réutilisables à volonté
- Les fonctions permettent la décomposition des algorithmes en sous-problèmes (*raffinements successifs*)
- Prennent en entrée des paramètres
- Restituent à l'algorithme appelant un ou plusieurs résultats
- Définies par :
  - Un en-tête :
    - › *Nom* : identifiant de la fonction
    - › *Liste des paramètres* : informations extérieures à la fonction
    - › *Résultat* : valeur de retour de la fonction
    - › *Description* en langage naturel du rôle de la fonction
  - Un corps :
    - › Algorithme de la fonction

## En-tête de fonction

- Repéré par le mot clé **fonction**
- Exemple :

```
fonction NomFonction( mode nomParam : typeParam, ... ) : ret typeRet
/* indique ce que fait la fonction et le rôle de ses paramètres */
```

  - NomFonction : identifiant de la fonction (idem vars avec *majuscule au début*)
  - mode : à donner pour chaque paramètre
    - › in : paramètre *donnée*, non modifiable par la fonction
    - › out : paramètre *résultat*, modifié par la fonction, valeur initiale non utilisée
    - › in-out : paramètre *donnée/résultat*, valeur initiale utilisée, modifié par la fonction
  - nomParam : identifiant du paramètre dans la fonction
  - typeParam : type du paramètre
  - retour :
    - › Mot clé **ret** suivi du type de la valeur renvoyée par la fonction (typeRet)
    - › Si pas de retour, le type de retour est le mot clé **vide**

## Corps de la fonction

- Construit comme un algorithme classique :
  - Idée de l'algorithme
  - Lexique **local** des constantes
  - Lexique **local** des variables
  - Algorithme de nomFonction
- Les variables/constantes définies dans une fonction sont utilisables **uniquement** dans cette fonction et **ne peuvent pas** être utilisées en dehors de celle-ci (**variables locales** ou **constantes locales**)
- Elles sont **détruites** dès que l'on sort de la fonction

# Déclaration / Définition / Retour

- La *déclaration* des fonctions est placée avant le lexique des variables dans le *lexique des fonctions* (liste des en-têtes)
- La *définition* des fonctions est placée après l'algorithme principal dans la section *définition des fonctions* (corps des fonctions)
- Lorsque la fonction a un *retour*, sa dernière instruction doit être `retour_de_NomFonction ← valeurDeRetour`  
Cela permet de renvoyer effectivement une valeur à l'algorithme appelant
- Les fonctions sans retour sont appelées *procédures*

## Exemple

fonction VolPrisme( in côté : réel, in hauteur : réel ) : ret réel  
/\* calcule le volume d'un prisme de *côté* et *hauteur* donnés \*/

Lexique local des variables :

hauteurTri	(réel)	Hauteur du triangle formant la base	INTERMÉDIAIRE
surfTri	(réel)	Surface du triangle formant la base	INTERMÉDIAIRE

Algorithme de VolPrisme :

```
hauteurTri ← côté * sqrt(3) / 2
surfTri ← côté * hauteurTri / 2
retour_de_VolPrisme ← surfTri * hauteur
```

# Paramètres formels / effectifs

- ⚠ les paramètres de la fonction *ne doivent pas* être redéfinis dans le lexique local des variables de la fonction
- Ce sont déjà des variables/constantes *locales* à la fonction dans lesquelles on recopie les éléments reçus de l'algorithme appelant
- Les paramètres dans l'en-tête de la fonction sont les *paramètres formels*
- Les paramètres utilisés lors de l'appel de la fonction par l'algorithme appelant sont les *paramètres effectifs*
- Les fonctions peuvent ne pas avoir de paramètres

## Exemple (suite)

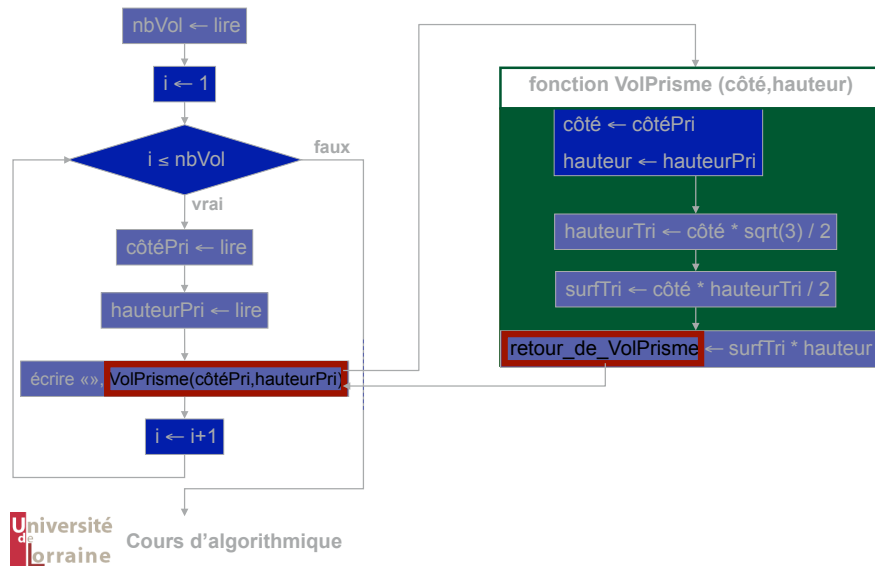
Lexique des variables :

nbVol	(entier)	Nombre de volumes à calculer	DONNÉE
i	(entier)	Compteur de la boucle	INTERMÉDIAIRE
côtéPri	(réel)	Côté d'un prisme	DONNÉE
hauteurPri	(réel)	Hauteur d'un prisme	DONNÉE

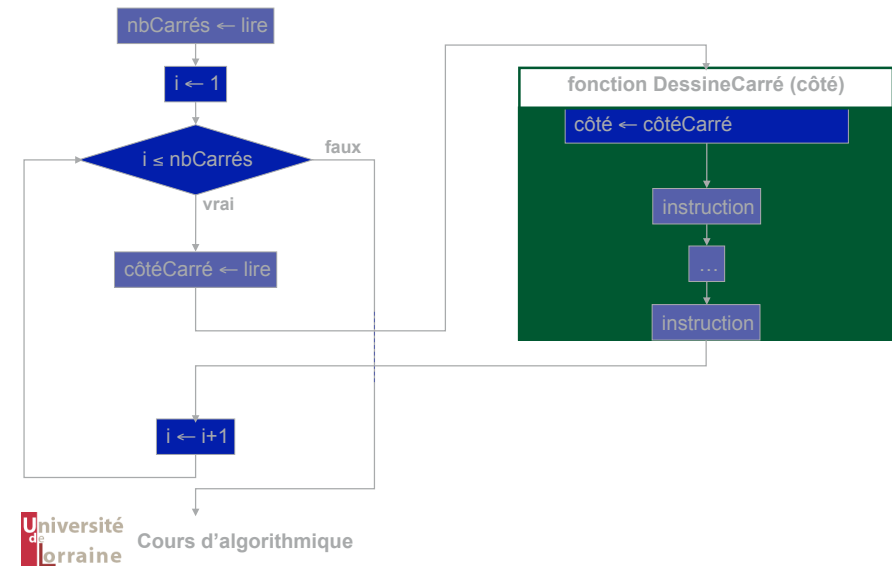
Algorithme :

```
nbVol ← lire
pour i de 1 à nbVol
  côtéPri ← lire
  hauteurPri ← lire
  écrire "Le volume du prisme est ", VolPrisme(côtéPri, hauteurPri)
fpour
```

## Explication



## Fonction sans retour



## Fonctions à résultats multiples

- Dans certains cas, une fonction peut générer plusieurs résultats
- Mais une fonction ne renvoie toujours qu'une valeur au plus !
- On communique les autres résultats via des paramètres modifiables (modes **out** ou **in-out**)
- Une bonne **convention** est de :
  - renvoyer un booléen indiquant si les résultats générés sont valides
  - placer tous les résultats dans des paramètres modifiables
  - si possible ne pas modifier les paramètres résultats si données invalides
- Exemple :
  - fonction PériSurf(in rayon:réel, **out** périm:réel, **out** surf:réel) : ret **booléen**  
/\* calcule le périmètre et la surface d'un disque de rayon donné, si le rayon n'est pas valide (<0), les paramètres ne sont pas modifiés et la fonction renvoie FAUX, sinon elle renvoie VRAI \*/

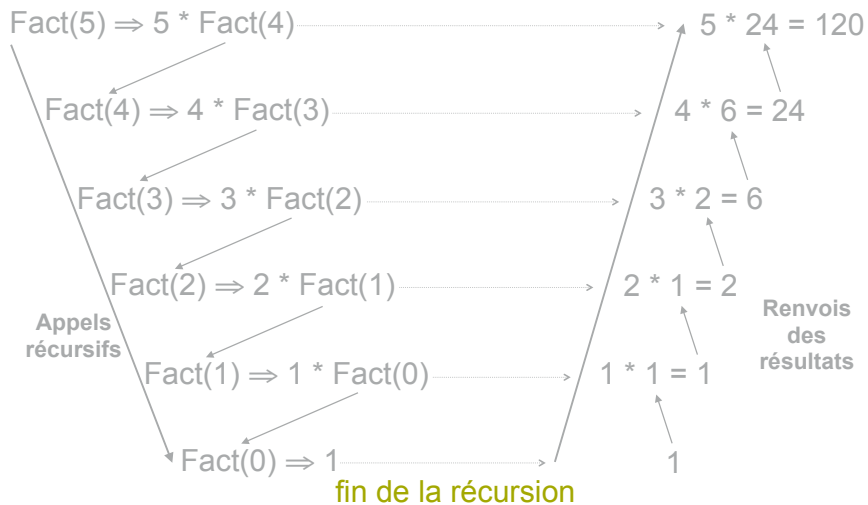
## Récursion

- Construire la solution d'un problème en utilisant la solution du **même** problème dans un contexte **différent** (plus simple)
- La suite des contextes doit tendre vers une solution directe (**cas terminal**)
- Adaptée à une certaine classe de problèmes
- Exemple : la factorielle  $\Rightarrow n! = n * (n-1)!$  et  $0! = 1$   
fonction Factorielle( in n : entier ) : ret entier  
/\* calcule la factorielle de n positif ou nul \*/  
Algorithme de Factorielle :

```

si n=0 alors
  retour_de_Factorielle ← 1
sinon
  retour_de_Factorielle ← n * Factorielle(n-1)
fsi
  
```

# Exemple d'exécution



# Caractéristiques

- Une fonction récursive doit contenir :
  - *au moins* un appel à elle-même avec des *paramètres différents*
  - *au moins* un cas où elle ne s'appelle pas
  - ⇒ *au moins* une conditionnelle pour séparer ces différents cas
- On dit que la récursivité est *terminale* lorsque l'appel récursif est la *dernière instruction* réalisée dans la fonction
- S'il y a des traitements *après* l'appel récursif, on a une récursivité *non terminale*

# Exemple

```

fonction LitEcritOrdo( in nb : entier ) : ret vide
/* lit et écrit dans le même ordre nb nombre entiers */
fonction LitEcritInv( in nb : entier ) : ret vide
/* lit et écrit dans l'ordre inverse nb nombre entiers */
    
```

Algorithme de LitEcritOrdo :

```

si nb > 0 alors
    valeur ← lire
    écrire valeur
    LitEcritOrdo(nb-1)
fsi
    
```

récursivité terminale

Algorithme de LitEcritInv :

```

si nb > 0 alors
    valeur ← lire
    LitEcritInv(nb-1)
    écrire valeur
fsi
    
```

récursivité non terminale

# Autre exemple

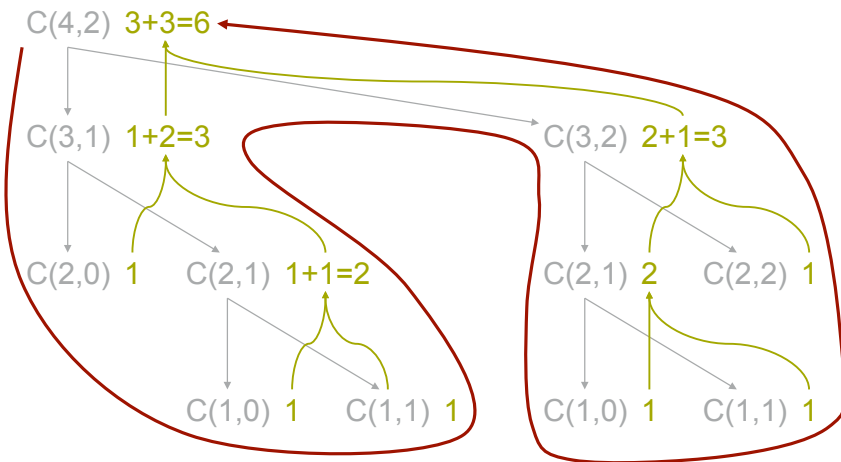
- Calcul de  $C(n,k) = C(n-1,k-1) + C(n-1,k)$  avec :
    - $C(n,n)=1, C(n,0)=1, C(n,k)=0$  quand  $k>n$
- fonction Cnk( in n : entier, in k : entier ) : ret entier  
 /\* ... \*/

Algorithme de Cnk :

```

si n < k alors
    retour_de_Cnk ← 0
sinon
    si n = k ou k = 0 alors
        retour_de_Cnk ← 1
    sinon
        retour_de_Cnk ← Cnk(n-1, k-1) + Cnk(n-1, k)
    fsi
fsi
    
```

# Exécution



# Les tableaux

- Structure de données qui permet de rassembler un ensemble de valeurs de *même* type sous un *même* nom en les différenciant par un *indice*

tNotes  un tableau de 6 réels

- Déclaration :

tNotes	(réel tableau[6])	Liste des notes	...
--------	-------------------	-----------------	-----

Type des éléments du tableau

Nombre d'éléments dans le tableau

# Les tableaux

- Structure de données qui permet de rassembler un ensemble de valeurs de *même* type sous un *même* nom en les différenciant par un *indice*

tNotes  un tableau de 6 réels

- Déclaration :

tNotes	(réel tableau[6])	Liste des notes	...
--------	-------------------	-----------------	-----

- Chaque élément est repéré dans le tableau par un indice entre 0 à *taille-1* (tNotes[-1] et tNotes[6] *ne sont pas valides* !)

0 1 2 3 4 5



on accède à la case 2 par tNotes[2]

⚠ c'est la 3ème case !

# Tableaux de constantes

- Placé dans le lexique des constantes
- On indique le contenu par la liste des valeurs entre { }
- On garde le formalisme des noms de constantes

- Exemple :

- On a besoin de la liste des nombres de jours des mois de l'année  
⇒ on peut définir un tableau dans le lexique des constantes

T_JOURS_MOIS	(entier tableau[12])	={31,28,31,30,31,30,31,31,30,31,30,31}	...
--------------	----------------------	--	-----

# Initialisation d'un tableau

- Lorsque l'on connaît les valeurs initiales à placer dans un tableau, on peut le faire en une seule instruction :  
tableau ← { liste\_valeurs }
- Le **nombre de valeurs** dans la liste doit correspondre au **nombre d'éléments** du tableau
- Exemple :
  - on a un tableau de 8 entiers tVals, on peut l'initialiser par :  
tVals ← { 5, 9, -4, 2, 12, 17, 2, 7 }

# Lecture/affichage d'un tableau

- Pour initialiser un tableau avec des valeurs fournies par l'utilisateur, on est obligé de lire les valeurs **une par une**
- Une **boucle** de lecture est donc nécessaire

Algorithme :

```
pour i de 0 à nbElem-1
    tab[i] ← lire
fpour
```

Attention à ne pas sortir du tableau !

- L'affichage se fait aussi élément par élément

Algorithme

```
pour i de 0 à nbElem-1
    écrire tab[i]
fpour
```

# Tableau et fonction

- On peut passer un tableau en paramètre d'une fonction
- Il faut en général passer aussi la **taille** du tableau
- Exemple :

fonction LirePrix(out tPrix : réel tableau, in taille : entier) : ret vide

/\* lit **taille** prix et les stocke dans **tPrix** \*/

Lexique local des variables :

i	(entier)	Indice de la boucle de lecture	INTERMÉDIAIRE
---	----------	--------------------------------	---------------

Algorithme de LirePrix :

```
pour i de 0 à taille-1
    tPrix[i] ← lire
fpour
```

# Exemple

- Édition d'une facture correspondant à l'achat de produits en quantités données parmi une liste de NB\_PROD produits de prix des donnés et numérotés de 1 à NB\_PROD

lexique des constantes :

NB_PROD	(entier) = 10	Nombre de produits à vendre
---------	---------------	-----------------------------

lexique des fonctions :

fonction LirePrix(out tPrix : réel tableau, in taille : entier) : ret vide

lexique des variables :

tPrix	(réel tableau[NB_PROD])	Liste des prix des produits	DONNÉE
total	(réel)	Montant total de la facture	RÉSULTAT
numProd	(entier)	Suite des numéros de produits achetés	DONNÉE
quantité	(entier)	Suite des quantités de produits	DONNÉE

# Exemple (suite)

Algorithme :

```
LirePrix(tPrix,NB_PROD) /* remplissage de la liste des prix des
                          produits */
total ← 0
numProd ← lire          /* lecture du premier numéro de produit */
tant que ¬_échecLecture_
  si numProd ≥ 1 et numProd ≤ NB_PROD alors
    quantité ← lire
    total ← total + quantité * tPrix[numProd-1]
  fsi
  numProd ← lire
ftant
écrire " le montant total est de ", total, " € "
```

Vérification de la validité du numéro

Décalage pour les indices du tableau

# Parcours en sens inverse

- On peut aussi parcourir un tableau de la fin vers le début

Algorithme :

```
pour i de 0 à taille-1
  tab[i] ← lire
fpour
pour i de taille-1 à 0 en descendant
  écrire tab[i]
fpour
```

⇒ impression de la liste des valeurs lues en ordre inverse

# Compter les voyelles dans un texte

Résultat :

Afficher le nombre d'occurrences des lettres 'a','e','i','o','u' dans un texte

Idée :

- Lire le texte caractère par caractère
- Compter les occurrences des voyelles
- Afficher les nombres d'occurrences des voyelles

Lexique des constantes :

NB_VOY	(entier) = 5	Nombre de voyelles dans l'alphabet
--------	--------------	------------------------------------

Lexique des variables :

tNbVoy	(entier tableau[NB_VOY])	Tableau des compteurs	RÉSULTAT
carLu	(caractère)	Suite des caractères du texte lu	DONNÉE

# Algorithme

Algorithme :

```
tNbVoy ← {0,0,0,0,0}
carLu ← lire
tant que ¬_échecLecture_
  selon que carLu est
    'a' : tNbVoy[0] ← tNbVoy[0] + 1
    'e' : tNbVoy[1] ← tNbVoy[1] + 1
    'i' : tNbVoy[2] ← tNbVoy[2] + 1
    'o' : tNbVoy[3] ← tNbVoy[3] + 1
    'u' : tNbVoy[4] ← tNbVoy[4] + 1
  fselon
  carLu ← lire
ftant
écrire "a : ", tNbVoy[0], "e : ", tNbVoy[1], ...
```

# Avec un tableau de constantes

Lexique des constantes :

NB_VOY	(entier)	=5	Nombre de voyelles dans l'alphabet
T_VOY	(caractère tableau[NB_VOY])	={'a','e','i','o','u'}	Liste des voyelles

Algorithme :

```

tNbVoy ← {0,0,0,0,0}      ...
carLu ← lire              pour i de 0 à NB_VOY-1
tant que !_échecLecture_  écrire T_VOY[i], tNbVoy[i]
    pour i de 0 à NB_VOY-1  fpour
        si carLu=T_VOY[i] alors
            tNbVoy[i] ← tNbVoy[i]+1
        fsi
    fpour
carLu ← lire
ftant
...
    
```

# Les caractères

- Notés entre apostrophes :
  - Exemples : 'a', 'A', '3', '{' sont des caractères
- On utilise un seul jeu de caractères à la fois
- Il suit certaines conventions :
  - Il contient tous les caractères utilisables (lettres, chiffres, ponctuations)
  - Chaque caractère est repéré par sa *position dans le jeu de caractères* ⇒ *Notion d'ordre* entre les caractères
  - Les chiffres sont contigus et dans l'ordre
  - Les lettres minuscules sont contiguës et dans l'ordre
  - Les lettres majuscules sont contiguës et dans l'ordre
  - L'ordre entre ces différents sous-ensembles peut être quelconque

# Opérations sur les caractères

- Addition/soustraction d'un entier :  $car \pm entier \Rightarrow car$ 
  - Effectue un *décalage* dans le jeu de caractères, du nombre de positions spécifié à partir du caractère donné :
    - › Vers la droite avec l'addition
    - › Vers la gauche avec la soustraction

Exemples : 'a'+1 ⇒ 'b', '5'-3 ⇒ '2', 'G'+0 ⇒ 'G'

										0	1	2	3	4	5	6	7	8	9		
										G	H	I	J	K	L	M	N	O	P	Q	
R	S	T	U	V	W	X	Y	Z							a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z		

# Opérations sur les caractères

- Différence entre deux caractères :  $car - car \Rightarrow entier$ 
    - Renvoie le *décalage relatif* entre les deux caractères : le déplacement nécessaire pour arriver au 1er car en partant du 2nd
- Exemples : 'b'-'a' ⇒ 1, '3'-'5' ⇒ -2, 'G'-'G' ⇒ 0, 'A'-'a' ...

										0	1	2	3	4	5	6	7	8	9							
										A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z													a	b	c	d	e	f
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z							

# Opérations sur les caractères

- Comparaisons : `car < = > ≤ ≥ ≠ car ⇒ booléen`
  - Compare deux caractères selon leurs positions dans le jeu de caractères
  - Un caractère est *inférieur* à un autre s'il est placé *avant*
  - Exemples : 'a' < 'b' ⇒ vrai, '5' > '9' ⇒ faux, 'G' ≠ 'l' ⇒ vrai

										0	1	2	3	4	5	6	7	8	9	
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	
R	S	T	U	V	W	X	Y	Z						a	b	c	d	e	f	
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	

- Il n'est pas nécessaire de connaître les positions absolues des caractères dans la table

# Les chaînes de caractères

- Tableau de caractères se différenciant par son contenu :
  - La chaîne contient au moins une occurrence du caractère spécial '\0'
  - Ce caractère marque la fin de la chaîne
- Exemple : la chaîne "bonjour" correspond à

b	o	n	j	o	u	r	\0													
---	---	---	---	---	---	---	----	--	--	--	--	--	--	--	--	--	--	--	--	--

- Conséquences :
  - Il faut un tableau d'au moins *N+1 cases* pour stocker une chaîne de *N caractères* (à cause du '\0')
  - Lors du parcours d'une chaîne, il n'est pas nécessaire de connaître sa taille car on s'arrête *dès que* l'on rencontre le caractère '\0'
  - Le type *chaîne* est utilisé pour marquer la différence sémantique avec les tableaux de caractères

# Déclaration / Lecture

- Déclaration :

maChaîne	(chaîne[20])	Une chaîne de 19 caractères au plus	...
----------	--------------	-------------------------------------	-----

- Lecture d'une chaîne :
  - directement avec l'instruction `lire`
  - Exemple : `maChaîne ← lire`
  - Les caractères sont rangés dans la chaîne dans l'ordre de lecture
  - Le '\0' est ajouté automatiquement *juste après* le dernier caractère lu
- La lecture d'une chaîne s'arrête sur le premier séparateur rencontré :
  - espace* ou *retour à la ligne*

⚠ La chaîne doit pouvoir contenir ce qui est lu !  
Une vérification est souvent nécessaire

# Remarque

⚠ Le type *chaîne* est différent du type *caractère tableau* :

nom	(chaîne[20])	Nom de l'étudiant	...
prénom	(caractère tableau[20])	Prénom de l'étudiant	...

- `nom` est terminé par un '\0' et sa lecture et son affichage sont directs
- `prénom` ne contient pas de '\0' et doit être lu/affiché caractère par caractère

Exemple :

```

nom ← lire           /* 19 caractères au plus */
i ← 0
carLu ← lire
tant que ¬_écheclecture_ et carLu ≠ ' ' et carLu ≠ '\n'
    et i < 20
    prénom[i] ← carLu /* 20 caractères au plus */
    i ← i + 1
    carLu ← lire
ftant
    
```

# Exécution

```

nom ← lire
i ← 0
carLu ← lire
tant que ...
    prénom[i] ← carLu
    i ← i + 1
    carLu ← lire
ftant
    
```

nom =				
S h o w \0				
i=0				
carLu='A'				
Vrai	Vrai	Vrai	Vrai	Faux
A	A r	A r t	A r t y	
i=1	i=2	i=3	i=4	
'r'	't'	'y'	'\n'	

# Manipulation de chaînes

- Plusieurs opérations sont possibles sur les chaînes :
  - Déterminer la longueur d'une chaîne
  - Copier une chaîne dans une autre
  - Concaténer deux chaînes (l'une à la suite de l'autre)
  - Comparer deux chaînes (ordre lexicographique)
  - Rechercher une sous-chaîne dans une chaîne
- La longueur des chaînes n'est pas explicitement nécessaire pour effectuer ces opérations (marqueur de fin '\0')
- L'affectation d'une chaîne (sauf lecture) ne peut se faire que caractère par caractère

# Recherche dans un tableau

- Trouver l'indice d'un élément particulier dans le tableau
  - Une première solution est de parcourir tout le tableau et de s'arrêter dès que l'on trouve la valeur cherchée
- ⚠ Il faut prévoir de s'arrêter à la fin du tableau si la valeur n'est pas présente

```

fonction Recherche( in tab: entier tableau, in taille : entier, in val : entier ) : ret entier
/* renvoie l'indice de val dans tab de taille taille si elle y est et -1 sinon */
    
```

Lexique local des variables :

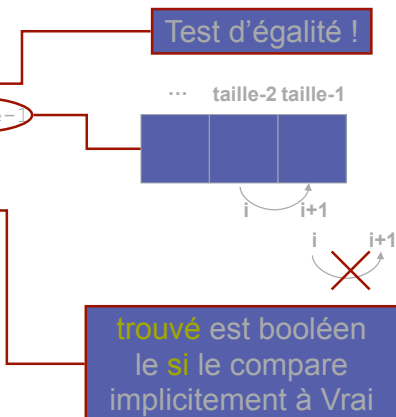
i	(entier)	Suite des indices des cases du tableau	INTER / RÉS
trouvé	(booléen)	VRAI si val est dans tab et faux sinon	INTERMÉDIAIRE

# Algorithme simple

Algorithme de Recherche :

```

i ← 0
trouvé ← tab[i]=val
tant que ¬trouvé et i<taille-1
    i ← i + 1
    trouvé ← tab[i]=val
ftant
si trouvé alors
    retour_de_Recherche ← i
sinon
    retour_de_Recherche ← -1
fsi
    
```



# Recherche dans un sous-tableau

fonction RechercheDans( in tab : entier tableau, in deb : entier, in taille : entier, in val : entier ) : ret entier  
 /\* renvoie l'indice de val dans tab entre les indices deb et deb+taille-1 et -1 sinon \*/



Algorithme de RechercheDans :

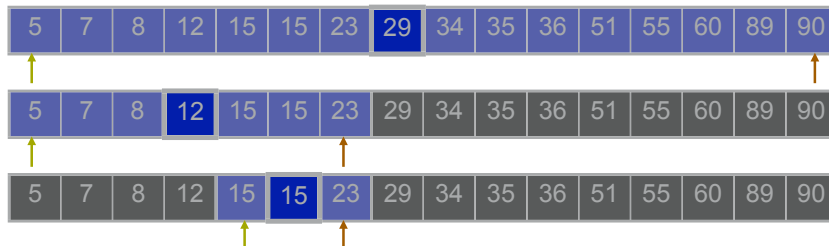
```

i ← deb
trouvé ← tab[i]=val
tant que ¬trouvé et i<deb+taille-1
    i ← i + 1
    trouvé ← tab[i]=val
ftant
...
si trouvé alors
    retour_de_RechercheDans ← i
sinon
    retour_de_RechercheDans ← -1
fsi
    
```

# Recherche dichotomique

- Arrêt de la recherche :
  - Lorsque l'on trouve l'élément recherché
  - Lorsque l'on arrive sur un intervalle de recherche vide (mêmes indices de début et de fin de l'intervalle)
- En général, on choisit l'élément du milieu pour la séparation

Exemple : recherche de 15



# Recherche dichotomique

- Ne s'applique qu'à des tableaux *triés* dont on connaît le *sens* de tri
- Principe : réduire l'intervalle de recherche à chaque étape
  - Choisir un élément dans le tableau :  $t[i]$ 

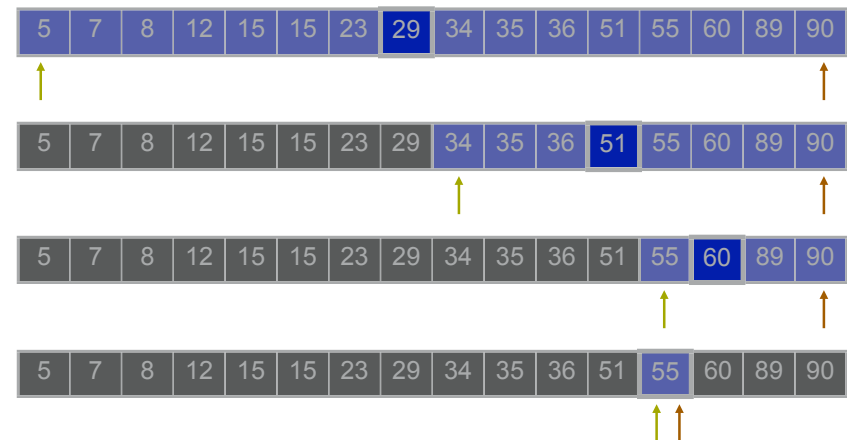
Éléments  $\leq t[i]$ 
 $t[i]$ 
Éléments  $\geq t[i]$
  - Si la valeur recherchée est  $> t[i]$ , on continue la recherche dans la partie des éléments  $\geq t[i]$ 

Éléments  $\leq t[i]$ 
 $t[i]$ 
Éléments  $\geq t[i]$
  - Sinon, si la valeur recherchée est  $< t[i]$ , on continue dans la partie des éléments  $\leq t[i]$ 

Éléments  $\leq t[i]$ 
 $t[i]$ 
Éléments  $\geq t[i]$
  - Enfin, lorsque la valeur est égale à  $t[i]$ , on a trouvé !!

# Exemple

- Recherche de 52 :



# Écriture récursive

```
fonction RechDicho( in tab : entier tableau, in deb : entier, in fin : entier,  
                  in val : entier ) : ret entier
```

```
/* renvoie l'indice de val dans tab entre deb et fin si elle y est et -1 sinon */
```

Idée de l'algorithme :

- Vérifier si l'on est sur la case recherchée ou que l'on a terminé la recherche
- Si ce n'est pas le cas, regarder de quel côté peut se trouver la valeur
  - Recommencer la recherche sur le nouvel intervalle correspondant
- Si on a terminé, on renvoie l'indice si c'est la bonne case ou -1 sinon

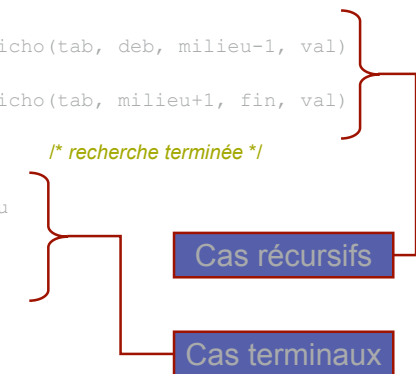
Lexique local des variables :

milieu	(entier)	Milieu de l'intervalle de recherche	INTERMÉDIAIRE
--------	----------	-------------------------------------	---------------

# Algorithme récursif

Algorithme de rechDicho :

```
milieu ← (deb+fin)/2  
si tab[milieu] ≠ val et deb<fin alors /* recherche non terminée */  
  si val<tab[milieu] alors  
    retour_de_RechDicho ← RechDicho(tab, deb, milieu-1, val)  
  sinon  
    retour_de_RechDicho ← RechDicho(tab, milieu+1, fin, val)  
fsi  
sinon /* recherche terminée */  
  si tab[milieu] = val alors  
    retour_de_RechDicho ← milieu  
  sinon  
    retour_de_RechDicho ← -1  
fsi  
fsi
```



# Écriture itérative

```
fonction RechDicho2( in tab : entier tableau, in deb : entier, in fin : entier,  
                   in val : entier ) : ret entier
```

```
/* renvoie l'indice de val dans tab entre deb et fin si elle y est et -1 sinon */
```

Idée de l'algorithme :

- Vérifier si l'on est sur la case recherchée ou que l'on a terminé la recherche
- Si ce n'est pas le cas, regarder de quel côté peut se trouver la valeur
  - Recommencer la recherche sur le nouvel intervalle correspondant
- Si on a terminé, on renvoie l'indice si c'est la bonne case ou -1 sinon

Lexique local des variables :

milieu	(entier)	Milieu de l'intervalle de recherche	INTERMÉDIAIRE
min	(entier)	Début de l'espace de recherche	INTERMÉDIAIRE
max	(entier)	Fin de l'espace de recherche	INTERMÉDIAIRE

# Algorithme itératif

Algorithme de RechDicho2 :

```
milieu ← (deb+fin)/2  
min ← deb  
max ← fin  
tant que tab[milieu] ≠ val et min<max  
  /* recherche non terminée */  
  si val<tab[milieu] alors  
    max ← milieu-1  
  sinon  
    min ← milieu+1  
  fsi  
  milieu ← (min+max)/2  
ftant  
/* recherche terminée */  
si tab[milieu] = val alors  
  ... /* même cas terminaux que version récursive */  
fsi
```

# Tableaux à 2 dimensions

- Représentation de matrices, tables, images...
- Un tableau contient des éléments de même type :
  - Le type peut être quelconque et donc aussi un tableau

⚠ un tableau à 2 dimensions ≡ un tableau de tableaux

Exemple : `table (entier tableau[7] tableau[4])`

définit `table` comme un tableau de 4 cases contenant chacune un tableau de 7 entiers

- Représentation *à plat* de la variable `table` :



## Exemple (1/4)

- On veut stocker et manipuler les notes des étudiants.
- ➔ Utilisation d'un tableau 2D contenant :
  - Un étudiant par ligne
  - Une matière par colonne
  - Dimensions NBE x NBM fixées : NBE et NBM définies en constantes

Tableau `tNotes[NBE][NBM]`

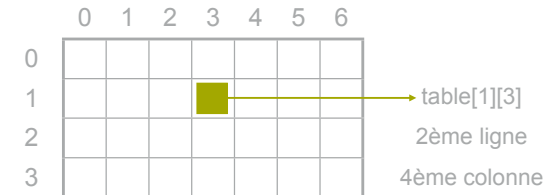
	Matière 1 0	...	Matière nbM NBM-1
Étudiant 1 0		...	
...	...	...	...
Étudiant nbE NBE-1		...	

# Représentation matricielle

- Déclaration :

<code>table</code>	<code>(entier tableau[4][7])</code>	Tableau de 4 lignes et 7 colonnes	...
--------------------	-------------------------------------	-----------------------------------	-----

- Représentation *matricielle* de table :



- Accès à un élément :

- `NomDuTableau[ indice 1ère dimension ][ indice 2ème dimension ]`  
 entre 0 et `tailleDim1-1`    entre 0 et `tailleDim2-1`

## Exemple (2/4)

- On utilise deux tableaux de chaînes de caractères pour les noms des matières et les noms d'étudiants

<code>tEtu</code>	<code>(chaîne[30] tableau[NBE])</code>	Liste des étudiants	...
<code>tMat</code>	<code>(chaîne[30] tableau[NBM])</code>	Liste des matières	...

- Fonctions de remplissage des tableaux :

fonction LireListe(out tab : chaîne tableau, in nbElem : entier) : ret vide  
 /\* lit `nbElem` chaînes et les place dans `tab` \*/

fonction LireNotes(out tNotes : réel tableau tableau, in tEtu : chaîne tableau,  
 in tMat : chaîne tableau, in nbE : entier, in nbM : entier  
 ) : ret vide

/\* lit `nbE x nbM` notes et les place dans `tNotes` \*/

# Fonction LireListe

fonction LireListe(out tab : chaîne tableau, in nbElem : entier) : ret vide  
 /\* lit nbElem chaînes et les place dans tab \*/

Lexique local des variables :

i	(entier)	Compteur de la boucle de lecture	INTERMÉDIAIRE
---	----------	----------------------------------	---------------

Algorithme de LireListe :

```
pour i de 0 à nbElem-1
    tab[i] ← lire
fpour
```

- Cette fonction est utilisable pour la lecture des noms des étudiants et la lecture des noms des matières

# Fonction LireNotes

fonction LireNotes(out tNotes : réel tableau tableau, ... , in nbM : entier) : ret vide  
 /\* lit nbE x nbM notes et les place dans tNotes \*/

Lexique local des variables :

iEtu	(entier)	Indice de l'étudiant	INTERMÉDIAIRE
iMat	(entier)	Indice de la matière	INTERMÉDIAIRE

Algorithme de LireNotes :

```
pour iEtu de 0 à nbE-1
    pour iMat de 0 à nbM-1
        répéter
            écrire "Entrez la note de ", tEtu[iEtu], " en ", tMat[iMat]
            tNotes[iEtu][iMat] ← lire
        tant que tNotes[iEtu][iMat]<0 ou tNotes[iEtu][iMat]>20
    fpour
fpour
```

# Calcul des moyennes

Résultat :

Afficher les moyennes par étudiant et les moyennes par matière

Lexique des constantes :

NBE	(entier)	=110	Nombre d'étudiants
NBM	(entier)	=15	Nombre de matières

Lexique des variables :

iEtu	(entier)	Indice de l'étudiant courant	INTERMÉDIAIRE
iMat	(entier)	Indice de la matière courante	INTERMÉDIAIRE
moyEtu	(réel)	Moyenne des notes par étudiant	RÉSULTAT
moyMat	(réel)	Moyenne des notes par matière	RÉSULTAT
tMat	(chaîne[30] tableau[NBM])	Liste des noms des matières	DONNÉE
tEtu	(chaîne[30] tableau[NBE])	Liste des noms des étudiants	DONNÉE
tNotes	(réel tableau[NBE][NBM])	Notes des étudiants	DONNÉE

# Calcul des moyennes

Algorithme :

```
LireListe(tMat, NBM)
LireListe(tEtu, NBE)
LireNotes(tNotes, tEtu, tMat, NBE, NBM)
...
pour iEtu de 0 à NBE-1
    moyEtu ← 0
    pour iMat de 0 à NBM-1
        moyMat ← 0
        pour iEtu de 0 à NBE-1
            moyMat ← moyMat
                + tNotes[iEtu][iMat]
        fpour
        moyMat ← moyMat/NBE
        écrire tMat[iMat], moyMat
    fpour
...
```

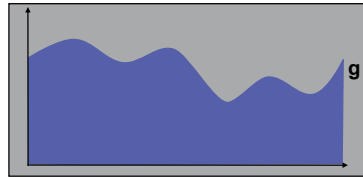
L'ordre des indices du tableau ne change pas !

L'ordre des boucles change !

# Complexité algorithmique

- Les modèles de complexité sont nécessaires pour comparer plusieurs algorithmes résolvant un même problème
- La complexité est principalement utilisée pour :
  - Montrer l'**optimalité** d'un algorithme
  - Déterminer la **complexité minimale** d'un problème
- Comparaisons asymptotiques de fonctions ( $f$  et  $g$ ):
  - $f$  est **bornée supérieurement** par  $g$  si  $\forall x \in \mathcal{D}, f(x) \leq g(x)$
  - $f$  admet  $g$  comme borne supérieure **asymptotique** si  $\exists x_0 \in \mathcal{D} \text{ t.q. } \forall x \geq x_0$
  - On a logiquement les notations symétriques  $f(x) \leq g(x) \quad (f \preceq g)$
- Pour une fonction  $g$ , on définit les trois ensembles suivants :

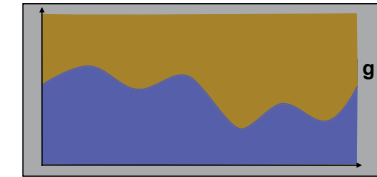
$$O(g) = \{f \mid \exists k > 0 : f \leq k \times g\}$$



# Complexité algorithmique

- Les modèles de complexité sont nécessaires pour comparer plusieurs algorithmes résolvant un même problème
- La complexité est principalement utilisée pour :
  - Montrer l'**optimalité** d'un algorithme
  - Déterminer la **complexité minimale** d'un problème
- Comparaisons asymptotiques de fonctions ( $f$  et  $g$ ):
  - $f$  est **bornée supérieurement** par  $g$  si  $\forall x \in \mathcal{D}, f(x) \leq g(x)$
  - $f$  admet  $g$  comme borne supérieure **asymptotique** si  $\exists x_0 \in \mathcal{D} \text{ t.q. } \forall x \geq x_0$
  - On a logiquement les notations symétriques  $f(x) \leq g(x) \quad (f \preceq g)$
- Pour une fonction  $g$ , on définit les trois ensembles suivants :

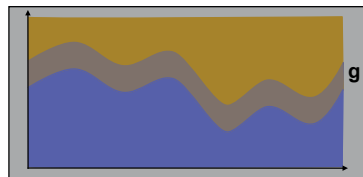
$$\Omega(g) = \{f \mid \exists k > 0 : f \geq k \times g\}$$



# Complexité algorithmique

- Les modèles de complexité sont nécessaires pour comparer plusieurs algorithmes résolvant un même problème
- La complexité est principalement utilisée pour :
  - Montrer l'**optimalité** d'un algorithme
  - Déterminer la **complexité minimale** d'un problème
- Comparaisons asymptotiques de fonctions ( $f$  et  $g$ ):
  - $f$  est **bornée supérieurement** par  $g$  si  $\forall x \in \mathcal{D}, f(x) \leq g(x)$
  - $f$  admet  $g$  comme borne supérieure **asymptotique** si  $\exists x_0 \in \mathcal{D} \text{ t.q. } \forall x \geq x_0$
  - On a logiquement les notations symétriques  $f(x) \leq g(x) \quad (f \preceq g)$
- Pour une fonction  $g$ , on définit les trois ensembles suivants :

$$\Theta(g) = O(g) \cap \Omega(g)$$



# Règles de complexité

- On calcule toujours une complexité **relativement à une mesure particulière** de l'algorithme (nb ops, nb tests, mém,...)
- La complexité est généralement directement associée à l'algorithme lorsqu'il s'agit du **nombre d'opérations**
- Mais il est essentiel de toujours **préciser ce que l'on mesure**
- **Quelques règles :**
  - Variables pour les trois ensembles :
    - $O(a \cdot f(n) + b) = O(f(n))$  pour  $a$  et  $b$  des constantes réelles
    - $O(f(n) + g(n)) = O(f(n))$  si  $g$  est dans  $O(f(n))$
    - $O(f(n) \cdot g(n)) = O(f(n)) \cdot O(g(n))$
  - Pour tout polynôme  $p(n)$ , on a :
    - $O(\log^a p(n)) \subset O(n^b)$  pour tout réel  $a$  et tout réel  $b > 0$
    - $O(p(n)) \subset O(e^{a \cdot n})$  pour tout réel  $a > 0$

# Liste des complexités

- $O(1)$  : constante (indépendante de la taille des données)
- $O(\log_b(n))$  : logarithmique en base  $b$
- $O(n)$  : linéaire
- $O(n \log_b(n))$  : quasi-linéaire
- $O(n^2)$  : quadratique
- $O(n^p)$  : polynomiale
- $O(n^{\log_b(n)})$  : quasi-polynomiale
- $O(a^n)$  : exponentielle
- $O(n!)$  : factorielle

# Exemple comparatif

- On compare deux versions du calcul de  $x^y$  (puissance)

fonction Puiss1(in x : réel, y : entier) : ret réel				fonction Puiss2(in x : réel, y : entier) : ret réel			
i	(entier)	compteur de boucle	INTER	tmp	(réel)	résultat partiel de la puissance (demi puissance)	INTER
rés	(réel)	x à la puissance y	RÉSULTAT				
<pre> rés ← 1 pour i de 1 à y   rés ← rés * x fpour retour_de_Puiss1 ← rés                     </pre>				<pre> si y = 0 alors   retour_de_Puiss2 ← 1 sinon   si y = 1 alors     retour_de_Puiss2 ← x   sinon     tmp ← Puiss2(x, y/2)     si y%2 = 0 alors       retour_de_Puiss2 ← tmp*tmp     sinon       retour_de_Puiss2 ← tmp*tmp*x   fsi fsi                     </pre>			

# Exemple comparatif

- Évaluons les complexités des deux algorithmes en termes de multiplications (principale source de calculs ici) :
  - Puiss1 : 1 \* par itération et y itérations  $\Rightarrow O(y)$
  - Puiss2 : 1 ou 2 par appel récursif et  $\log_2(y)$  appels  $\Rightarrow O(\log_2 y)$
- Soient deux machines A et B t.q  $t_A = 1\text{ms}$  et  $t_B = 0,1\text{ms}$
- On calcule la puissance 1000 d'un réel en utilisant :
  - Puiss2 sur la machine A
  - Puiss1 sur la machine B
- Les temps d'exécutions seront approximativement :
  - $t_A(\text{Puiss2}) = \log_2 1000 \approx 10\text{ms}$  (entre 10 et 20ms pour être précis)
  - $t_B(\text{Puiss1}) = 1000 * 0,1 = 100\text{ms}$
- 5 à 10 fois plus rapide sur une machine 10 fois plus lente !!!

# Un jeu de morpion

- On veut écrire un algorithme qui gère une partie de morpion entre deux joueurs sur une grille de dimensions données :
  - Le but du jeu est d'aligner un certain nombre de pions identiques (ligne, colonne, diagonale)
  - Chaque joueur dispose d'un pion (motif) différent
  - Les joueurs jouent à tour de rôle
  - Le jeu s'arrête quand :
    - Un joueur a aligné 4 de ses pions
    - La grille est pleine (possibilité de match nul)
- Modélisation :
  - Grille : tableau 2D de caractères
  - Pions : caractères

# Résultat / Idée

## Résultat :

Affichage de la grille à chaque tour  
et affichage du résultat de la partie à la fin

## Idée de l'algorithme :

- Initialiser la grille du jeu (vide)
- Répéter les tours de jeu jusqu'à la fin de la partie :
  - Spécifier le joueur actif et le joueur en attente
  - Afficher la grille du jeu
  - Effectuer un tour de jeu avec le joueur actif
  - Vérifier si le joueur actif a gagné ou si la grille est pleine
- Afficher le résultat de la partie

# Lexiques

## Lexique des fonctions (suite) :

fonction Tour(in-out grille : car tableau tableau, in joueur : car,  
in nbL : entier, in nbC : entier, in cVide : car) : ret vide

/\* effectue un tour de jeu du joueur sur la grille \*/

fonction ChangerJoueur(in-out jActif:car, in-out jAttente:car) : ret vide

/\* échange les rôles entre joueur actif jActif et joueur en attente jAttente \*/

fonction Vainqueur(in grille : car tableau tableau, in joueur : car,  
in nbL : entier, in nbC : entier, in nbA : entier) : ret booléen

/\* retourne VRAI si la grille contient nbA pions joueur alignés, FAUX sinon \*/

# Lexiques

## Lexique des constantes :

HAUT	(entier)	=7	Hauteur de la grille
LARG	(entier)	=10	Largeur de la grille
NBA	(entier)	=4	Nombre de pions à aligner consécutivement pour le gain
VIDE	(caractère)	' '	Représente une case vide dans la grille

## Lexique des fonctions :

fonction InitGrille(out grille : car tableau tableau, in nbL : entier, in nbC : entier,  
in cVide : car) : ret vide

/\* initialise la grille de nbL lignes et nbC colonnes avec cVide \*/

fonction AfficherGrille(in grille: car tableau tableau, in nbL:entier,  
in nbC: entier) : ret vide

/\* affiche la grille du jeu \*/

# Lexiques

fonction GrillePleine(in grille : car tableau tableau, in nbL:entier,  
in nbC : entier, in cVide : car) : ret booléen

/\* retourne VRAI si la grille ne contient plus de case vide et FAUX sinon \*/

## Lexique des variables :

jActif	(caractère)	Motif du joueur actif	INTERMÉDIAIRE
jAttente	(caractère)	Motif du joueur en attente	INTERMÉDIAIRE
grille	(caractère tableau[HAUT][LARG])	Grille du jeu	RÉSULTAT
gagné	(booléen)	Devient VRAI lorsque un joueur a gagné	INTERMÉDIAIRE

# Algorithme principal

## Algorithme :

```
InitGrille(grille,HAUT,LARG,VIDE)
jActif ← 'O'
jAttente ← 'X'
répéter
  ChangerJoueur(jActif, jAttente)
  AfficherGrille(grille,HAUT,LARG)
  Tour(grille, jActif, HAUT, LARG,VIDE)
  gagné ← Vainqueur(grille, jActif, HAUT, LARG, NBA)
  tant que ¬gagné et ¬GrillePleine(grille, HAUT, LARG, VIDE)
  si gagné alors
    écrire "Le joueur avec les pions ",jActif," a gagné "
  sinon
    écrire "Match nul !"
fsi
```

# Fonction AfficherGrille

fonction AfficherGrille(in grille: car tableau tableau, in nbL: entier,  
in nbC: entier) : ret vide

## Lexique local des variables :

ligne	(entier)	indice de ligne	INTERMÉDIAIRE
colonne	(entier)	indice de colonne	INTERMÉDIAIRE

## Algorithme de AfficherGrille :

```
pour ligne de 0 à nbL-1
  pour colonne de 0 à nbC-1
    écrire grille[ligne][colonne]," "
  fpour
  écrire "\n"
fpour
```

# Fonction InitGrille

fonction InitGrille(out grille : car tableau tableau, in nbL : entier,  
in nbC : entier, in cVide : car ) : ret vide

## Lexique local des variables :

ligne	(entier)	indice de ligne	INTERMÉDIAIRE
colonne	(entier)	indice de colonne	INTERMÉDIAIRE

## Algorithme de InitGrille :

```
pour ligne de 0 à nbL-1
  pour colonne de 0 à nbC-1
    grille[ligne][colonne] ← cVide
  fpour
```

# Fonction ChangerJoueur

fonction ChangerJoueur(in-out jActif : car, in-out jAttente : car) : ret vide

## Lexique local des variables :

inter	(caractère)	variable d'échange	INTERMÉDIAIRE
-------	-------------	--------------------	---------------

## Algorithme de ChangerJoueur :

```
inter ← jActif
jActif ← jAttente
jAttente ← inter
```

# Fonction GrillePleine

fonction GrillePleine(in grille : car tableau tableau, in nbL : entier,  
in nbC : entier, in cVide : car) : ret booléen

Lexique local des variables :

ligne	(entier)	indice de ligne	INTERMÉDIAIRE
colonne	(entier)	indice de colonne	INTERMÉDIAIRE
res	(booléen)	VRAI si la grille est pleine	RÉSULTAT

Algorithme de GrillePleine :

```
ligne ← 0
res ← VRAI
tant que ligne < nbL et res
  colonne ← 0
  tant que colonne < nbC et res
    res ← grille[ligne][colonne]
    retour_de_GrillePleine ← res
    #cVide
  fin tant
  ligne ← ligne + 1
fin tant
```

# Remarque sur le test de grille pleine

- Le test de grille pleine est relativement coûteux puisqu'il implique de parcourir potentiellement toute la grille
- On peut éviter ce parcours si on compte le nombre de coups joués par les deux joueurs :
  - On ne peut pas jouer plus de coups qu'il y a de cases dans le tableau
- Le test de grille pleine revient donc à tester si le nombre de coups joués est égal au nombre de cases de la grille, soit  $NBL \times NBC$
- Il faut introduire une variable *nbCoups* dans l'algorithme principal et l'incrémenter de 1 à chaque tour de jeu

# Fonction Tour

fonction Tour(in-out grille : car tableau tableau, in joueur : car,  
in nbL : entier, in nbC : entier, in cVide : car) : ret vide

Lexique local des variables :

lig	(entier)	Numéro de ligne où le joueur place son pion	DONNÉE
col	(entier)	Numéro de colonne où le joueur place son pion	DONNÉE

Algorithme de Tour :

```
répéter
  tant que grille[lig-1][col-1] ≠ cVide
    grille[lig-1][col-1] ← joueur
  fin tant
  lig ← lire
  tant que lig < 1 ou lig > nbL
    répéter
      col ← lire
      tant que col < 1 ou col > nbC
        ...
```

On peut tester la case car les indices recalés sont valides

# Fonction Vainqueur

fonction Vainqueur(in grille : car tableau tableau, in joueur : car,  
in nbL : entier, in nbC : entier, in nbA : entier) : ret booléen

Lexique local des fonctions :

fonction TestL(in grille : car tableau tableau, in numL : entier,  
in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

/\* renvoie VRAI si la ligne numL contient nbA motifs joueur consécutifs \*/

fonction TestC(in grille : car tableau tableau, in nbL : entier,  
in numC : entier, in joueur : car, in nbA : entier) : ret booléen

/\* renvoie VRAI si la colonne numC contient nbA motifs joueur consécutifs \*/

fonction TestDiag(in grille : car tableau tableau, in nbL : entier,  
in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

/\* renvoie VRAI si au moins un des deux types de diagonales contient nbA motifs joueur consécutifs \*/

# Fonction Vainqueur

Lexique local des variables :

ind	(entier)	Indice de ligne et de colonne	INTERMÉDIAIRE
gagné	(booléen)	Vrai si on a nbA motifs joueur en ligne ou colonne	RÉSULTAT

Algorithme de Vainqueur :

```

gagné ← FAUX
ind ← 0
tant que ¬gagné et ind < nbL
    gagné ← TestL(grille, ind, nbC,
                 joueur, nbA)
    ind ← ind + 1
ftant
ind ← 0
...
tant que ¬gagné et ind < nbC
    gagné ← TestC(grille, nbL, ind,
                 joueur, nbA)
    ind ← ind + 1
ftant
si gagné alors
    retour_de_Vainqueur ← VRAI
sinon
    retour_de_Vainqueur ← TestDiag(grille,
                                   nbL, nbC, joueur, nbA)
fsi
    
```

# Fonction TestL

fonction TestL(in grille : car tableau tableau, in numL : entier, in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

Problème : y-a-t-il une séquence de nbA éléments dans la ligne ?



Principe : On utilise un compteur relatif au motif

- On parcourt la ligne
  - Lorsque la case contient le motif, on incrémente le compteur
  - Lorsque la case ne contient pas le motif, on remet le compteur à zéro
- ⇒ Compte le nombre de motifs consécutifs
- Si ce nombre atteint le seuil voulu, on a gagné !

Lexique local des variables :

ind	(entier)	Indice de colonne	INTERMÉDIAIRE
nbMotifs	(entier)	Nombre de motifs joueur consécutifs	INTERMÉDIAIRE

# Fonction TestL

Algorithme de TestL :

```

ind ← 0
nbMotifs ← 0
tant que ind < nbC et nbMotifs < nbA
    si grille[numL][ind]=joueur alors
        nbMotifs ← nbMotifs + 1
    sinon
        nbMotifs ← 0
    fsi
    ind ← ind + 1
ftant
retour_de_TestL ← nbMotifs=nbA
    
```

Parcours des colonnes de la ligne numL

On s'arrête dès que l'on a trouvé ce que l'on cherche

On a trouvé lorsque nbMotifs est égal à nbA

# Fonction TestC

fonction TestC(in grille : car tableau tableau, in nbL : entier, in numC : entier, in joueur : car, in nbA : entier) : ret booléen

Même principe que TestL mais en parcourant une colonne



Lexique local des variables :

ind	(entier)	Indice de ligne	INTERMÉDIAIRE
nbMotifs	(entier)	Nombre de motifs joueur consécutifs	INTERMÉDIAIRE

# Fonction TestC

Algorithme de TestC :

```
ind ← 0
nbMotifs ← 0
tant que ind < nbL et nbMotifs < nbA
  si grille[ind][numC]=joueur alors
    nbMotifs ← nbMotifs + 1
  sinon
    nbMotifs ← 0
  fsi
  ind ← ind + 1
ftant
retour_de_TestC ← nbMotifs=nbA
```

Boucle sur les lignes  
La colonne est constante

# Fonction TestDiag

fonction TestDiag(in grille : car tableau tableau, in nbL : entier, in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

Lexique local des fonctions :

fonction DiagHB(in grille : car tableau tableau, in nbL : entier, in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

/\* teste les diagonales dans le sens haut-gauche bas-droit \*/



fonction DiagBH(in grille : car tableau tableau, in nbL : entier, in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

/\* teste les diagonales dans le sens bas-gauche haut-droit \*/



# Fonction TestDiag

Algorithme de TestDiag :

```
si DiagHB(grille,nbL,nbC,joueur,nbA) alors
  retour_de_TestDiag ← VRAI
sinon
  retour_de_TestDiag ← DiagBH(grille,nbL,nbC,joueur,nbA)
fsi
```

# Fonction DiagHB

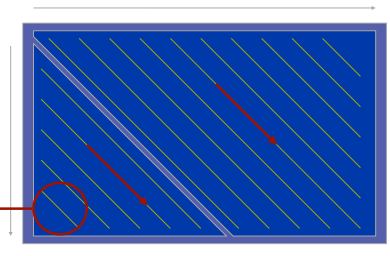
fonction DiagHB(in grille : car tableau tableau, in nbL : entier, in nbC : entier, in joueur : car, in nbA : entier) : ret booléen

Lexique local des variables :

lig	(entier)	Indice de ligne	INTERMÉDIAIRE
col	(entier)	Indice de colonne	INTERMÉDIAIRE
nbMotifs	(entier)	Nombre de motifs <b>joueur</b> consécutifs	INTERMÉDIAIRE

2 fois  
2 boucles imbriquées

Longueur = nbA



# Fonction DiagHB

Algorithme :

```

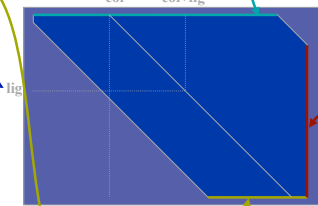
nbMotifs ← 0          /* initialisation */
parcours et traitement de la partie supérieure
si nbMotifs < nbA alors
  nbMotifs ← 0        /* changement de zone donc remise à zéro
                       pour ne pas continuer le dernier
                       comptage de la zone précédente */
  parcours et traitement de la partie inférieure
fsi
retour_de_DiagHB ← nbMotifs=nbA
  
```

# Fonction DiagHB

Algorithme : (1ère boucle imbriquée, partie supérieure)

```

col ← 0
tant que col < nbC - nbA et nbMotifs < nbA
  // Parcours de la diagonale issue de la colonne col
  lig ← 0
  tant que lig < nbL et col + lig < nbC et nbMotifs < nbA
    si grille[lig][col + lig] = joueur alors
      nbMotifs ← nbMotifs + 1
    sinon
      nbMotifs ← 0
    fsi
    lig ← lig + 1
  ftant
  col ← col + 1
ftant
  
```

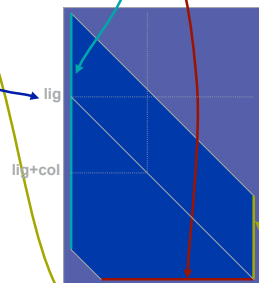


# Fonction DiagHB

Algorithme : (2ème boucle imbriquée, partie inférieure)

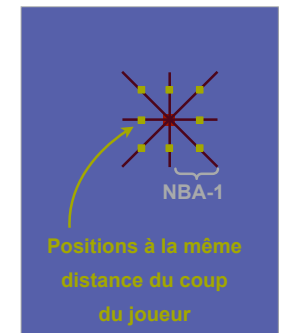
```

lig ← 1 // ligne 0 traitée dans la 1ère boucle imbriquée
tant que lig < nbL - nbA et nbMotifs < nbA
  // Parcours de la diagonale issue de la ligne lig
  col ← 0
  tant que col < nbC et lig + col < nbL et nbMotifs < nbA
    si grille[lig + col][col] = joueur alors
      nbMotifs ← nbMotifs + 1
    sinon
      nbMotifs ← 0
    fsi
    col ← col + 1
  ftant
  lig ← lig + 1
ftant
  
```



# Version minimale de Vainqueur

- Après le coup d'un joueur, il n'est pas nécessaire de tester toute la grille !
- Si l'on connaît la position jouée par le joueur au dernier coup
  - Il suffit de tester si ce dernier coup forme un segment de NBA pions au moins
- On peut donc ne parcourir que les NBA-1 positions autour du coup dans les 4 directions possibles d'alignement
- On suppose la position du dernier coup joué stockée dans les variables lig et col (entiers)
- On peut compter selon les 4 directions à la fois en utilisant un tableau de 4 compteurs des motifs consécutifs
  - tNbMotifs (entier tableau[4])
- Il faut également un tableau de 8 coefs (0|1) pour distinguer les comptages consécutifs selon chaque branche
  - tCoefSens (entier tableau[8])
- Idée :
  - On part d'un motif (celui joué)
  - On parcourt la suite des distances au coup joué
    - On teste les 8 positions à cette distance (les points jaunes sur la figure)



# Nouvel algorithme de Vainqueur

```

gagné ← FAUX // On démarre dans un contexte où pas encore de victoire
tNbMotifs ← {1,1,1,1} // On part du coup joué pour chaque direction
tCoefSens ← {1,1,1,1,1,1,1,1} // Chaque branche est prise en compte au départ
dist ← 1 // On commence le parcours aux voisins directs du coup
tant que ¬gagné et dist < nba
  // Boucles imbriquées pour traiter les 8 positions à la distance dist du coup
  pour sigL de -1 à 1 // On pourrait remplacer par des boucles conditionnelles
    pour sigC de -1 à 1 // pour s'arrêter dès que l'on gagne
      lVois ← lig+sigL*dist // Ligne de la position courante à tester
      cVois ← col+sigC*dist // Colonne de la position courante à tester
      si sigL≠0 ou sigC≠0 et (lVois≥0 et lVois<nbl et cVois≥0 et cVois<nbc alors
        numCoef ← 3*(sigL+1)+sigC+1-(3*(sigL+1)+sigC+1) // valeurs entre 0 et 7 inclus
        si grille[lVois][cVois]=joueur alors // valeurs entre 0 et 3 inclus
          numDir ← abs(3*sigL+sigC) - 1
          tNbMotifs[numDir] ← tNbMotifs[numDir] + tCoefSens[numCoef]
          si tNbMotifs[numDir] = nba alors
            gagné ← VRAI
            fsi
          sinon
            tCoefSens[numCoef] ← 0 // On ne comptabilise plus
            fsi
        fpour
      fpour
    fpour
  dist ← dist + 1
ftant
retour_de_Vainquer ← gagné

```

Évite la position centrale

Assure que la position du voisin parcouru est dans la grille de jeu

Formules permettant de distinguer les 4 directions possibles et les 8 branches parcourues avec des valeurs compatibles avec les indices des tableaux

On pourrait minimiser les calculs en insérant quelques tests supplémentaires

# Tri sur les vecteurs

- La notion d'ordre n'est pas implicite dans un tableau
- Il est souvent nécessaire d'avoir des valeurs triées
  - Recherche dichotomique
- Nous allons voir les tris en ordre croissant suivants :
  - Tri par insertion
  - Tri par sélection-permutation
  - Tri à bulles
  - Double tri à bulles
- On peut facilement déduire les versions décroissantes
- Ces tris *simples* sont *peu efficaces* sur de grands vecteurs
- On utilise alors les tris :
  - Fusion
  - Rapide (quick sort)



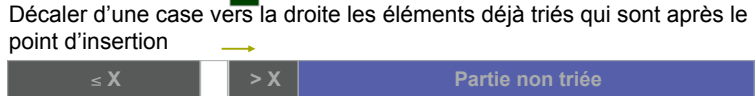

# Tri par insertion

- Utilisé pour trier les cartes que l'on a en main
- Principe :
  - On prend un élément et on recherche sa place dans la partie déjà triée puis on l'insère à cet endroit, et ainsi de suite
- Application aux tableaux :
  - Deux parties dans le tableau :
 


Partie triée	Partie non triée
--------------	------------------
  - On procède itérativement en prenant à chaque fois un élément de la partie non triée et en l'insérant à la *bonne place* dans la partie triée
 

⇒ La proportion de partie triée/partie non triée va donc évoluer
  - À chaque fois, on prend le 1er élément de la partie non triée comme élément à insérer dans la partie déjà triée

# Étapes d'insertion d'un élément

- L'insertion de l'élément courant se fait en 4 étapes :
  - Trouver le point d'insertion dans la partie triée
 
  - Recopier l'élément courant
 
  - Décaler d'une case vers la droite les éléments déjà triés qui sont après le point d'insertion
 
  - Placer l'élément courant à l'emplacement ainsi libéré
 

# Remarques

- On voit que la **nouvelle** partie grisée est **toujours triée**  

- Au départ, on a logiquement :
  - Partie triée : vide
  - Partie non triée : tout le tableau
- Mais on peut gagner une étape en prenant :
  - Partie triée : 1er élément du tableau
  - Partie non triée : le reste du tableau
- On termine quand la **partie non triée** devient **vide** : quand le **dernier élément** du tableau a été **inséré** dans la partie triée

# Algorithme

fonction TriInsert(in-out vect : réel tableau, in taille : entier) : ret vide  
/\* trie le tableau vect de taille éléments par insertion \*/


Lexique local des variables :

valeur	(réel)	Valeur à insérer dans la partie triée	INTERMÉDIAIRE
limite	(entier)	Indice du 1er élément de la partie non triée	INTERMÉDIAIRE
place	(entier)	Indice d'insertion de la valeur dans la partie triée	INTERMÉDIAIRE
trouvée	(booléen)	Vrai lorsque la place d'insertion est trouvée	INTERMÉDIAIRE

# Algorithme

```
pour limite de 1 à taille-1 /*on commence à 1 car 1ère case déjà triée */
  valeur ← vect[limite]
  place ← limite /*on parcourt la partie triée depuis sa fin */
  trouvée ← vect[place-1] ≤ valeur /*on a trouvé qd on arrive sur un élt ≤ */
  tant que ¬trouvée
    vect[place] ← vect[place-1] /*on décale vect[place-1] d'une case
    à droite car elle est > à la valeur à placer */
    place ← place - 1 /*on passe à la case précédente dans la partie triée */
    si place>0 alors /*si on n'est pas au début du tableau : */
      trouvée ← vect[place-1] ≤ valeur /*on continue la recherche */
    sinon
      trouvée ← VRAI /*on arrête la recherche (dernière place possible) */
  fsi
  vect[place] ← valeur /*on place la valeur à l'emplacement trouvé */
fpour
```

# Tri par sélection-permutation

- On aimerait **ne plus** avoir à **modifier** la partie déjà triée (déplacements des éléments trop coûteux)
  - On distingue toujours les deux parties des éléments mais cette fois :  

  - Ajout d'un nouvel élément **à la fin** de la partie triée
    - Il doit être ≤ aux éléments de la partie non triée
- Principe :
  - On cherche le plus petit élément dans la partie non triée (sélection) et on le place au début de cette partie (permutation)
- Nouvelle** zone triée = **ancienne** zone triée **plus** cet élément
- Nouvelle** zone non triée = **ancienne** zone non triée **moins** cet élément

# Étapes de sélection-permutation

- La sélection-permutation d'un élément se fait en 2 étapes :
  - Trouver le minimum dans la partie non triée



- Permuter avec le premier élément de la partie non triée



## Algorithme

fonction TriSélect(in-out vect : réel tableau, in taille : entier) : ret vide  
 /\* trie le tableau vect de taille éléments par sélection-permutation \*/

Lexique local des variables :

iMin	(entier)	Indice du minimum de la partie non triée	INTERMÉDIAIRE
min	(réel)	Valeur du minimum de la partie non triée	INTERMÉDIAIRE
limite	(entier)	Indice de la 1ère case de la partie non triée	INTERMÉDIAIRE
ind	(entier)	Indice de parcours du tableau pour trouver le min	INTERMÉDIAIRE

# Remarques

- On voit que la *nouvelle* partie grisée est *toujours triée*
- Au départ, on a logiquement :
  - Partie triée : vide
  - Partie non triée : tout le tableau
- Cette fois, on gagne une étape *à la fin* (dernier élément)
- On termine quand la *partie non triée* ne contient plus *qu'un seul* élément : il s'incorpore directement dans la partie triée



## Algorithme

```

pour limite de 0 à taille-2
    iMin ← limite
    /*on finit à l'avant dernière case */
    /*au départ, le min est initialisé avec
    la 1ère case de la partie non triée */

    pour ind de limite+1 à taille-1
        /*on recherche le min parmi toutes
        les valeurs suivantes jusqu'à la fin */

        si vect[ind]<vect[iMin] alors
            iMin ← ind
            /*on récupère l'indice du nouveau min */

        fsi

    fpour
    min ← vect[iMin]
    vect[iMin] ← vect[limite]
    vect[limite] ← min
    /*on échange le min trouvé */
    /*et 1ère case de la partie non triée */

fpour
    
```

# Tri à bulles

- Un parcours de tableau pour déplacer une seule valeur (le min)
- Coûteux ! On aimerait **déplacer plusieurs valeurs par parcours**
  - On distingue toujours la partie triée ( $\leq$ ) et la partie non triée :



- On fait descendre les éléments les plus petits par parcours successifs



- Principe :
  - Parcours droite-gauche de la partie non triée avec comparaison deux à deux des éléments consécutifs et remise en ordre éventuelle
- **Nouvelle** zone triée = **ancienne** zones triée **plus** élément min de la zone non triée

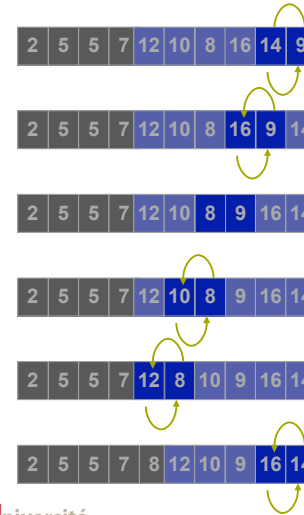
# Algorithme

fonction TriBulles(in-out vect : réel tableau, in taille : entier) : ret vide  
 /\* trie le tableau **vect** de **taille** éléments par insertion \*/

Lexique local des variables :

tmp	(réel)	variable pour l'échange des valeurs	INTERMÉDIAIRE
limite	(entier)	Indice de la 1ère case de la partie non triée	INTERMÉDIAIRE
ind	(entier)	Indice de parcours du tableau pour trouver le min	INTERMÉDIAIRE

# Déroulement d'un parcours



- On commence à **la fin** du tableau
- On compare les élt **deux à deux**
- On les **échange** s'ils ne sont pas dans l'ordre
- On recommence en se décalant d'une case à gauche jusqu'au début de la partie non triée
- À la fin du parcours, le **min** de la partie non triée est au **début** de celle-ci
- On recommence avec nouvelle zone non triée
- Arrêt quand zone non triée contient **un seul** élément ou est **vide**
- Remarque : le min n'est **pas la seule** valeur déplacée 9 a aussi été déplacé vers la gauche

# Algorithme

```

pour limite de 0 à taille-2 /*on finit à l'avant dernière case */
  pour ind de taille-1 à limite+1 en descendant
    /*on parcourt la zone non triée de droite à gauche :
      on s'arrête à limite+1 car on compare avec l'élément précédent */
    si vect[ind]<vect[ind-1] alors /*ordre inversé donc on échange */
      tmp ← vect[ind]
      vect[ind] ← vect[ind-1]
      vect[ind-1] ← tmp
    fsi
  fpour
fpour
    
```

# Remarques

- Variante du tri par sélection-permutation où la recherche du min est remplacée par les déplacements des petites valeurs
- Peut avoir un coût supérieur à cause des échanges (ordre inverse)
- Si au k<sup>ème</sup> parcours, la fin du tableau est déjà triée, on effectue quand même les taille-1-k parcours restants ⇒ **Pas efficace !**
- On s'arrête dès qu'il n'y a plus d'échange de valeurs
- On commence un nouveau parcours seulement si il y a eu au moins un échange dans le parcours précédent ⇒ Utilisation d'un booléen pour la détection d'échange ⇒ La boucle principale devient conditionnelle

# Algorithme

fonction TriBulles(in-out vect : réel tableau, in taille : entier) : ret vide  
/\* trie le tableau **vect** de **taille** éléments par insertion \*/

Lexique local des variables :

tmp	(réel)	variable pour l'échange des valeurs	INTERMÉDIAIRE
limite	(entier)	Indice de la 1 <sup>ère</sup> case de la partie non triée	INTERMÉDIAIRE
ind	(entier)	Indice de parcours du tableau pour trouver le min	INTERMÉDIAIRE
échange	(booléen)	VRAI si au moins un échange lors d'un parcours	INTERMÉDIAIRE

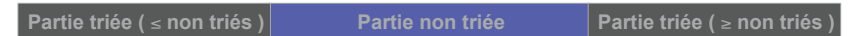
# Algorithme

```
limite ← 0 /*initialisation de la boucle tant que */
échange ← VRAI /*on le place à VRAI pour entrer dans la boucle */
tant que limite ≤ taille-2 et échange /*on continue si la partie non triée
    n'est pas vide ET s'il y a eu échange au parcours précédent */
    échange ← FAUX /*au début du parcours, il n'y a pas encore eu d'échange */
    pour ind de taille-1 à limite+1 en descendant
        si vect[ind]<vect[ind-1] alors
            tmp ← vect[ind]
            vect[ind] ← vect[ind-1]
            vect[ind-1] ← tmp
            échange ← VRAI /*on détecte un échange */
    fsi
    limite ← limite + 1
ftant
```

# Double tri à bulles

- Parcours dans les deux sens : bulles et poids

– On distingue trois parties cette fois :



– On fait descendre/monter les éléments les plus petits/grands par parcours successifs de la partie non triée

grandes valeurs se déplacent à droite → ← petites valeurs se déplacent à gauche

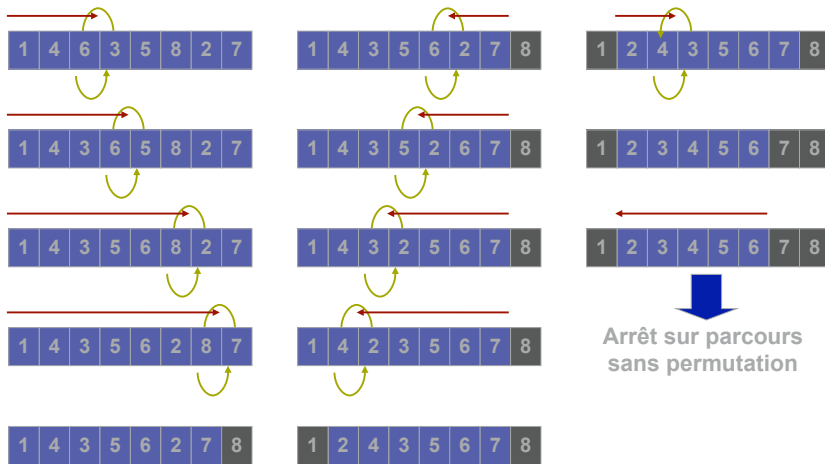


- Principe :

– Parcours de la partie non triée avec comparaison deux à deux des éléments consécutifs et remise en ordre éventuelle

- **Nouvelles** zones triées = **anciennes** zones triées **plus** éléments min/max de la zone non triée

# Exemple d'exécution



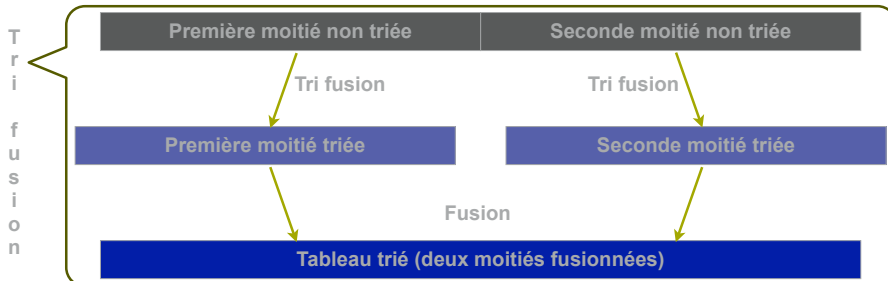
# Étude de complexité

- **Tri par insertion :**
  - Recherche et insertion d'un élément :  $O(n)$
  - Pour chaque élément de la partie non triée :  $O(n)$
  - Total :  $O(n^2)$
- **Tri par selection-permutation :**
  - Recherche du min dans la partie non triée :  $O(n)$
  - Permutation :  $O(1)$
  - Pour chaque élément de la partie non triée :  $O(n)$
  - Total :  $O(n^2)$
- **Tri à bulles :**
  - Une remontée de bulles :  $O(n)$
  - Combien de remontées nécessaires :  $O(n)$
  - Total :  $O(n^2)$
- **Double tri à bulles :**
  - Un parcours dans les deux sens :  $O(n)$
  - Nombre de parcours nécessaires :  $O(n)$
  - Total :  $O(n^2)$

En fait, le nombre d'échanges de voisins à faire est lié au nombre d'inversions d'ordre dans le tableau.  
Pour un ordre quelconque, ce nombre d'inversions est en moyenne  $n(n-1)/4 \Rightarrow O(n^2)$

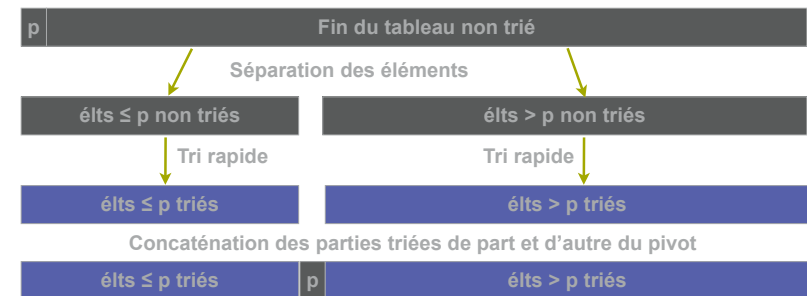
# Tri fusion

- Technique de **diviser pour régner** :
  - Principe similaire à la récursivité vu précédemment
  - Solution en fonction du même problème mais de taille inférieure
- Pour trier un tableau :
  - On trie la première moitié et la seconde moitié séparément
  - On fusionne les deux moitiés triées
- Récursion dans les tris des moitiés jusqu'à sous-parties suffisamment petites



# Tri rapide (quick sort)

- Également stratégie du **diviser pour régner** :
  - Choisir un élément *pivot*
  - Construire un tableau des éléments  $\leq$  pivot (tabInf)
  - Construire un tableau des éléments  $>$  pivot (tabSup)
  - Trier ces deux tableaux par tri rapide
  - Concaténer tabInf suivi du pivot suivi de tabSup
- Souvent, on prend le premier élément du tableau comme pivot



# Étude de complexité

- **Tri fusion :**
  - Fusion de deux sous-tableaux triés :  $O(n)$
  - Nombre de fusions à faire :  $O(\log_2(n))$
  - **Total :  $O(n \cdot \log_2 n)$**
- **Tri rapide :**
  - Séparation des éléments  $\leq$  et  $>$  au pivot :  $O(n)$
  - Nombre de séparations : ça dépend !
    - lorsque le pivot est le premier élément du tableau
      - si tableau trié ou presque  $O(n)$  séparations
      - sinon  $O(\log_2 n)$  séparations
    - lorsque le pivot est tiré aléatoirement dans l'intervalle de valeurs
      - le pire cas est encore en  $O(n)$  séparations
    - lorsque le pivot est la médiane (coupe le tableau en deux parties égales)
      - toujours  $O(\log_2 n)$  séparations
  - **Total :  $O(n \cdot \log_2 n)$**

# Petite comparaison

- On trie un tableau de 1 000 000 éléments sur deux machines différentes ayant un rapport de vitesse de 1000
  - Sur la machine plus rapide, on utilise un tri en  $O(n^2)$
  - Sur la machine plus lente, on utilise un tri en  $O(n \cdot \log_2 n)$
- Nombre d'opérations effectuées sur la machine rapide :
  - $O(n^2) = O(10^6 \times 10^6) = O(10^{12})$
- Nombre d'opérations effectuées sur la machine lente :
  - $O(n \cdot \log_2 n) = O(10^6 \times \log_2(10^6)) \approx O(10^6 \times 20) = O(2 \cdot 10^7)$
- Ratio du nombre d'opérations entre les deux :
  - $10^{12} / 2 \cdot 10^7 = 5 \cdot 10^4$
- **Machine rapide :  $10^{-9}$  s/op  $\Rightarrow$  temps du tri =  $10^3$  s (16 m 40 s)**
- **Machine lente :  $10^{-6}$  s/op  $\Rightarrow$  temps du tri = 20 s**
- On est 50 fois plus rapide sur une machine 1000 fois plus lente !

# Le type structure

- Les tableaux permettent de réunir plusieurs données de même type dans une seule entité
- La **structure** permet de réunir des **données de types quelconques dans une même entité** logique
- Les éléments de la structure (appelés **champs**) sont sensés avoir un lien logique entre eux :
  - ils représentent les différentes caractéristiques qui définissent l'entité
- Exemple : représentation d'un étudiant
  - Numéro d'étudiant (entier)
  - Nom (chaîne[30])
  - Prénom (chaîne[30])
  - Age (entier)

# Déclaration d'une variable structure

- Exemple : écriture de la structure étudiant  
Structure
  - num : (entier)
  - nom : (chaîne[30])
  - prénom : (chaîne[30])
  - age : (entier)
- Ainsi, la déclaration d'une variable *étudiant* peut se faire par :  
lexique des variables :

NOM	TYPE	DESCRIPTION	RÔLE
étudiant	(Structure numEtu : (entier) nom : (chaîne[30]) prénom : (chaîne[30]) age : (entier)	Les données d'un étudiant	...

# Définition d'un type structure

- Pour alléger l'écriture des définitions de variables, on peut aussi définir un nouveau type dans le *lexique des types* qui se place *avant* les autres lexiques
- La déclaration d'un nouveau type se fait simplement par :  
TypeNomDuType = définition du type
- Les noms des types que l'on construit commencent par *Type*
- Ainsi, la déclaration du type *TypeEtudiant* se fait par :

Lexique des types :

```
TypeEtudiant = structure
    num      : (entier)
    nom      : (chaîne[30])
    prénom   : (chaîne[30])
    age      : (entier)
```

# Type structure et accès

- La déclaration de la variable *étudiant* peut alors se faire par :  
Lexique des variables :

étudiant	(TypeEtudiant)	Les données d'un étudiant	...
----------	----------------	---------------------------	-----

- **Accès à un champ** : *.* entre la variable et le champ  
nomDeVariable.nomDuChamp

Exemple :

```
étudiant.nom ← lire
```

- **Affectation de structure** :

étudiant1 et étudiant2 sont 2 variables de type TypeEtudiant, l'instruction

```
étudiant1 ← étudiant2
```

*Recopie* les valeurs de *tous* les champs de la variable étudiant2 dans

étudiant1 : même effet que pour les types simples (entier, réel, ...)

# Structures et fonctions

- Une structure peut être passée en paramètre d'une fonction et aussi être renvoyée en résultat
- Exemples :

```
fonction LireEtu(out étu : TypeEtudiant) : ret booléen
```

```
/* lit les données d'un étudiant dans la structure étu et retourne VRAI si la lecture est correcte (pas de fin de saisie) */
```

```
TypeDate = structure
```

```
    jour      : (entier)
    mois      : (entier)
    année     : (entier)
```

```
fonction Demain(in aujourd'hui : TypeDate) : ret TypeDate
```

```
/* retourne une structure TypeDate contenant la date du lendemain */
```

# Exemple complet

Lexique des types :

```
TypeDate = structure
    jour : (entier)
    mois : (entier)
    année : (entier)
TypeEtudiant = structure
    num : (entier)
    nom : (chaîne[30])
    prénom : (chaîne[30])
    dateNais : (TypeDate)
```

Lexique des constantes :

NB_ETU	(entier)	=120	Nombre d'étudiant(e)s dans la promo
--------	----------	------	-------------------------------------

Lexique des fonctions :

```
fonction LireEtu(out étu : TypeEtudiant) : ret booléen
```

```
/* lit les données d'un étudiant et renvoie VRAI si lecture ok */
```

## Exemple (suite)

### Lexique des variables :

tabEtu	(TypeEtudiant tableau[NB_ETU])	Tableau des étudiants	INTER
i	(entier)	Indice de parcours du tableau	INTER
unEtu	(TypeEtudiant)	Étudiant(e) lu(e)	DONNÉE

### Algorithme :

```
i ← 0
tant que i < NB_ETU et LireEtu(unEtu)
  tabEtu[i] ← unEtu
  si tabEtu[i].dateNais.mois=6 alors
    écrire « l'étudiant numéro », unEtu.numEtu, « est né en Juin »
```

```
fsi
i ← i+1
ftant
```

...	<b>tabEtu[i]</b>	...
	num nom prénom dateNais jour mois année	

## Exemple (suite)

### Définition de la fonction LireEtu :

#### Lexique local des variables :

rés	(booléen)	VRAI si la lecture des données est valide	RÉSULTAT
-----	-----------	---	----------

#### Algorithme de lireEtu :

```
rés ← FAUX
étu.num ← lire
si ¬_échecLecture_ alors
  étu.nom ← lire
  si ...
    ... // imbrication des lectures, rés devient VRAI si toutes
    // les lectures jusqu'à la dernière sont effectives
  fsi
fsi
retour_de_LireEtu ← rés
```

## Liens avec le modèle objet

- Approche du cours :
  - Impérative hiérarchique : décompositions successives des étapes
  - Conception *basée sur les actions* à effectuer
- Approche objet :
  - Entités qui interagissent entre elles :
    - Modélisation des informations dans les entités (données)
    - Modélisation des actions possibles de chaque entité (méthodes)
  - Conception *basée sur les données* intervenant dans le problème
  - Les objets sont équivalents à des *structures* dont les *accès* sont *contrôlés* (parties privée et publique)
  - Assure la *cohérence* des informations à l'intérieur des objets
  - Les méthodes sont conçues avec l'approche impérative hiérarchique

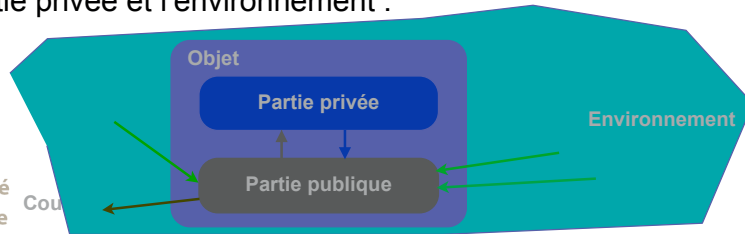
### Différence conceptuelle et méthodologique !

## Classe d'objets

- Les données manipulées sont des **objets**
- Chaque objet est une **instance** d'une **classe** (*un type d'objets*)
  - Exemple : la 2CV est une instance de la classe voiture
- Une classe est définie par :
  - **Un nom** :
    - l'identifiant de la classe (ex: voiture)
  - **Des attributs** :
    - les propriétés communes à tous les objet de la classe
    - ce sont les variables permettant de modéliser la classe d'objets
  - **Des méthodes** :
    - les fonctions agissant sur les objets de cette classe
    - l'**accès aux attributs** de la classe ne peut se faire que **par le biais de ses méthodes**
    - Contrainte visant à assurer la *cohérence* et la validité des attributs !

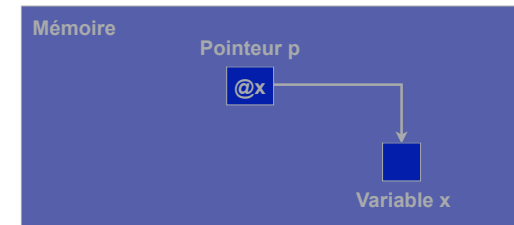
# Interface avec l'extérieur

- On peut considérer que les objets évoluent dans un environnement global
- L'objet et l'environnement sont deux choses *séparées*
- ➔ L'environnement n'a pas un accès direct à l'intérieur de l'objet (différence avec les structures impératives)
- Il doit passer par une méthode *publique* de l'objet, qui seule peut accéder à la partie *privée* de celui
- La partie publique d'un objet sert donc *d'interface* entre sa partie privée et l'environnement :



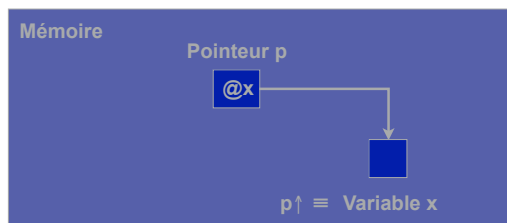
# Les pointeurs

- Une variable est définie par :
  - Un nom
  - Un type
  - Un emplacement mémoire réservé à une adresse précise
- Un pointeur sur un type  $T$  est une variable qui *contient* l'adresse mémoire d'une variable de type  $T$



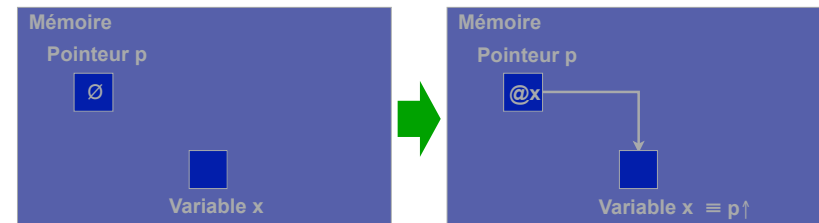
# Opérateurs élémentaires

- **Accès à la valeur pointée** :  $p \uparrow$   
↑ est l'opérateur *d'indirection* ou de *déréférencement*
- ⚠ le pointeur doit contenir une adresse *valide* sinon comportement indéfini
- **L'adresse d'une variable** est récupérée par :  $@x$   
@ est l'opérateur *d'adressage*



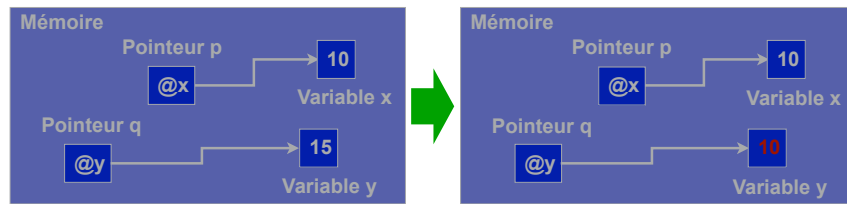
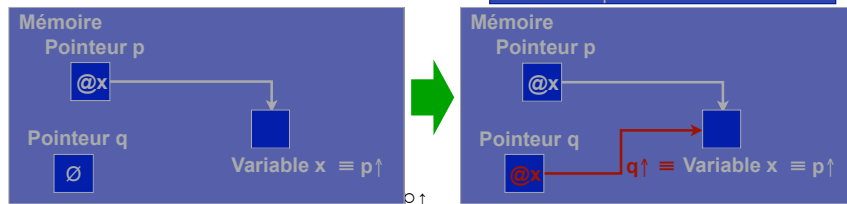
# Opérations sur les pointeurs

- **Initialisation d'un pointeur** :  $p \leftarrow \text{NIL}$ 
  - **NIL** indique que le pointeur pointe sur *rien*
  - Avant toute utilisation, il faut vérifier la valeur du pointeur
  - Dès qu'un pointeur n'est plus utilisé, il faut le remettre à NIL
- **Affectation** :  $p \leftarrow @x$ 
  - Place l'adresse de  $x$  dans  $p$



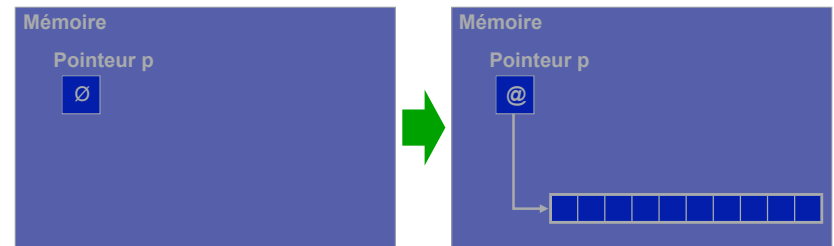
# Opérations sur les pointeurs

- Copie d'un pointeur :  $q \leftarrow p$



# Allocation dynamique

- Réserver un emplacement mémoire dynamiquement :
  - Créer un tableau de taille donnée pendant l'exécution de l'algorithme
  - Requiert le nombre et le type des éléments que l'on veut stocker
- Fonction d'allocation :  
fonction Alloue(in nb : entier, in typeElem : Type) : ret pointeur sur typeElem
- Retourne NIL si l'allocation n'est pas possible
- Exemple :  
`p ← Alloue(10,entier) // alloue un tableau de 10 entiers`



# Désallocation

- Lorsqu'un emplacement dynamique n'est plus utile, il faut le désallouer de façon à économiser la mémoire
- Fonction de désallocation :  
fonction Libère(in ptr : pointeur) : ret vide
- Après la libération mémoire, il est conseillé de réinitialisé le pointeur à Nil pour éviter des utilisations erronées
- Exemple :  
`Libère(p) // libère l'emplacement mémoire pointée par p mais NE modifie PAS p`  
`p ← NIL // permet de détecter que p ne pointe plus sur un emplacement valide`



# Exemple

Lexique des variables :

tab	(pointeur sur entier)	Tableau dynamique d'entiers	INTER
ind	(entier)	Indice du tableau	INTER
nb	(entier)	Nombre d'entiers dans le tableau	DONNÉE

Algorithme :

```

nb ← lire
si nb > 0 alors
    tab ← Alloue(nb,entier)
    si tab ≠ NIL alors // allocation réussie ! le tableau a bien été créé
        ... tab[ind] ... // utilisation de tab comme un tableau classique à nb éléments
                        // en fait la notation [ind] signifie « décalage de ind élts
                        // depuis le pointeur tab », donc depuis la première case
        Libère(tab) // libération du tableau dynamique après son utilisation
        tab ← NIL // réinitialisation du pointeur pour la suite de l'algo
    fsi
fsi
...
    
```

# Pointeurs et structures

- On peut définir des pointeurs sur n'importe quel type de données : entier, réel, tableau de ... mais aussi les **structures**
- Accès aux champs via un pointeur** : indirection suivie du `.`  
pointeur  $\uparrow$  `.nomDuChamp`

Exemple :

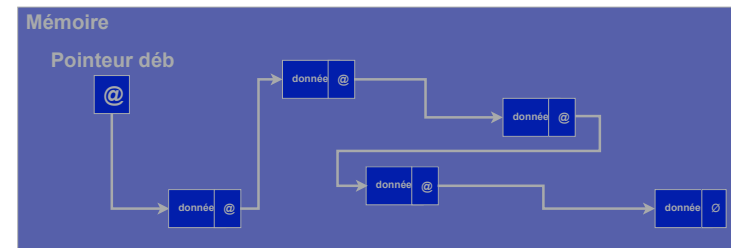
ptrEtu	(pointeur sur TypeEtudiant)	pointeur sur une structure d'étudiant	INTER
étudiant	(TypeEtudiant)	un étudiant	INTER

Algo :

```
ptrEtu ← @étudiant
ptrEtu↑.nom ← lire
...
écrire ptrEtu↑.dateNais.année
```

# Les listes chaînées

- Listes d'éléments (maillons) dont le nombre évolue **dynamiquement** pendant l'exécution de l'algorithme :
  - Les maillons sont de **même type**
  - Chaque maillon est relié au suivant par un pointeur
  - Le début de la liste est repéré par un pointeur sur le 1er maillon
  - La fin de la chaîne est repérée par la valeur NIL du pointeur sur le suivant dans le dernier maillon



# Opérations sur les listes chaînées

- Mise en place** :
  - Définition du maillon (créer un type qui est une structure)
  - Déclaration d'un pointeur sur le début de la liste et d'un pointeur de parcours
  - Construction de la liste :
    - Création d'un maillon (allocation dynamique)
    - Insertion d'un maillon dans la liste (au début, dans la liste, à la fin)
- Utilisation** :
  - Parcours de la liste
- Modification** :
  - Insertion d'un maillon dans la liste (idem ci-dessus)
  - Suppression d'un maillon dans la liste (au début, dans la liste, à la fin)
- Désallocation** :
  - Comme tout élément créé dynamiquement, il faut désallouer la liste :
    - Supprimer tous les éléments de la liste

# Définition d'un maillon

- On crée un **type** correspondant à **un maillon** de la liste
- Le maillon doit contenir les données à stocker ainsi qu'au moins un pointeur de chaînage sur un maillon de **même type**
  - Il faut utiliser une **structure**
- Il y a deux organisations possibles à l'intérieur du maillon :

Tous les éléments au <b>même</b> niveau	Données <b>séparées</b> des liaisons entre maillons
TypeMaillonPoint = structure	TypePoint = structure
lig : (entier)	lig : (entier)
col : (entier)	col : (entier)
coul : (entier)	coul : (entier)
svt : (pointeur sur TypeMaillonPoint)	TypeMaillonPoint = structure
	pt : (TypePoint)
	svt : (pointeur sur TypeMaillonPoint)

# Déclaration et initialisation

- Il faut déclarer un pointeur sur le début de la liste et au moins un pointeur supplémentaire pour les manipulations :

début	(pointeur sur TypeMaillonPoint)	Pointe sur le début de la liste	INTER
courant	(pointeur sur TypeMaillonPoint)	Pointe sur un élément	INTER

- Au début de l'algorithme, la liste est *vide* et *aucun maillon* n'est encore *alloué*
- La valeur à NIL permet de savoir si la liste est vide ou non
- Ainsi, il faut initialiser les pointeurs à NIL :

```
début ← NIL // la liste est vide
courant ← NIL // courant ne pointe sur rien
```

# Construction de la liste

- Pour construire la liste, il faut créer des maillons en utilisant l'allocation dynamique *d'un seul* élément à la fois :

```
courant ← Alloue(1, TypeMaillonPoint)
```

- Mais le maillon pointé par courant n'est ni *initialisé* ni *chaîné*
- L'initialisation des données se fait comme pour des variables classiques :

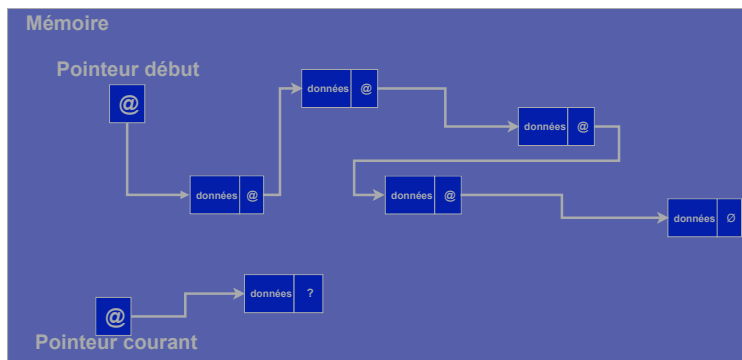
```
courant.col ← ... // affectation de la colonne du point courant
```

- Le *chaînage* consiste à insérer le nouveau maillon dans la liste en définissant :
  - Les liens sortant du maillon (un seul dans le cas le plus simple)
  - Les éléments qui pointent sur ce maillon (un seul en général)
  - On modifie toujours le maillon à insérer *avant* la liste

# Insertion d'un élément

- Au début de la liste :

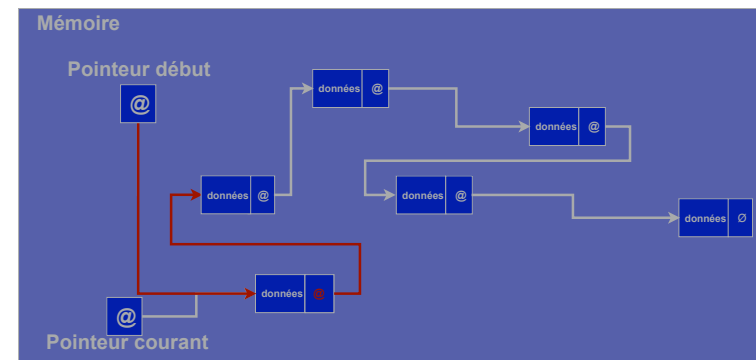
```
courant.svt ← début // le 1er maillon devient le suivant du futur 1er
début ← courant // le maillon courant devient le 1er maillon
```



# Insertion d'un élément

- Au début de la liste :

```
courant.svt ← début // le 1er maillon devient le suivant du futur 1er
début ← courant // le maillon courant devient le 1er maillon
```

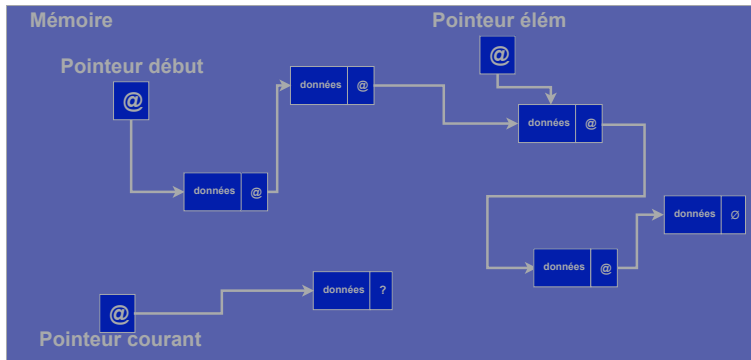


# Insertion d'un élément

- Dans la liste : entre un maillon (élément) et son suivant
 

```

courant↑.svt ← élém↑.svt // la liste n'est pas encore modifiée
élémt↑.svt ← courant // le maillon est inséré à la suite de élém
      
```

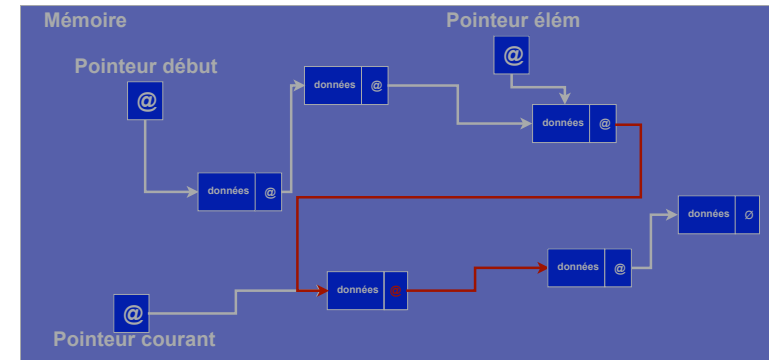


# Insertion d'un élément

- Dans la liste : entre un maillon (élément) et son suivant
 

```

courant↑.svt ← élém↑.svt // la liste n'est pas encore modifiée
élémt↑.svt ← courant // le maillon est inséré à la suite de élém
      
```

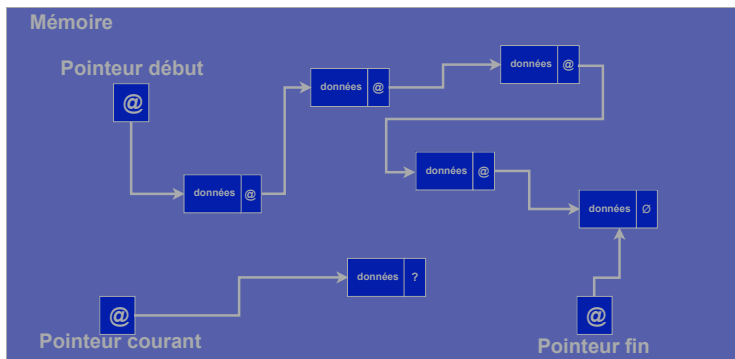


# Insertion d'un élément

- À la fin de la liste : idem mais entre le dernier maillon et rien
 

```

courant↑.svt ← fin↑.svt // revient à affecter courant↑.svt à NIL
// si la liste a été bien construite
fin↑.svt ← courant // le maillon courant devient le dernier élément
fin ← courant // MàJ du pointeur sur le dernier maillon si conservé
      
```

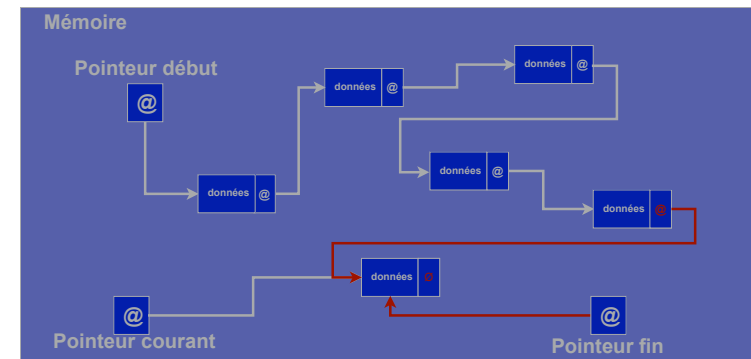


# Insertion d'un élément

- À la fin de la liste : idem mais entre le dernier maillon et rien
 

```

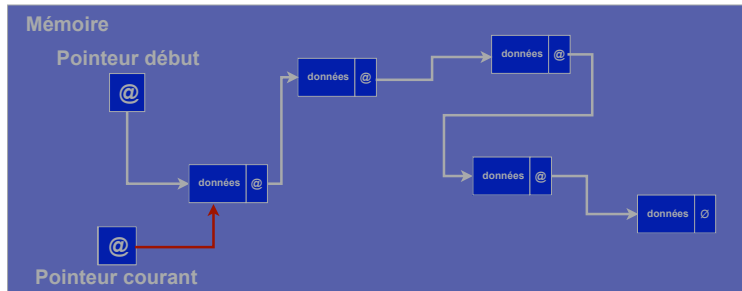
courant↑.svt ← fin↑.svt // revient à affecter courant↑.svt à NIL
// si la liste a été bien construite
fin↑.svt ← courant // le maillon courant devient le dernier élément
fin ← courant // MàJ du pointeur sur le dernier maillon si conservé
      
```



# Parcours de la liste

- Utilisation d'un pointeur de parcours :

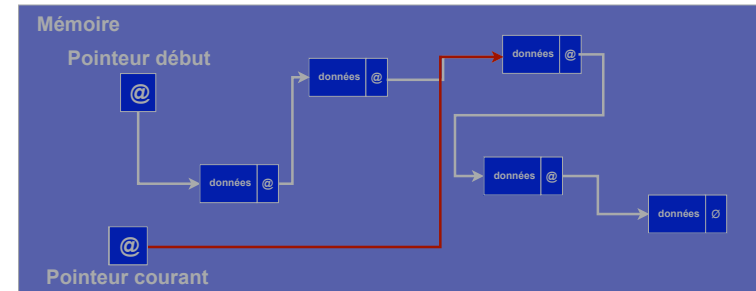
```
courant ← début // initialisation sur le début de la liste (si liste vide
// alors courant = NIL)
tant que courant ≠ NIL // on continue tant que l'on pointe sur un maillon
... courant; ... // traitement du maillon courant
courant ← courant;.svt // passage au maillon suivant
ftant
```



# Parcours de la liste

- Utilisation d'un pointeur de parcours :

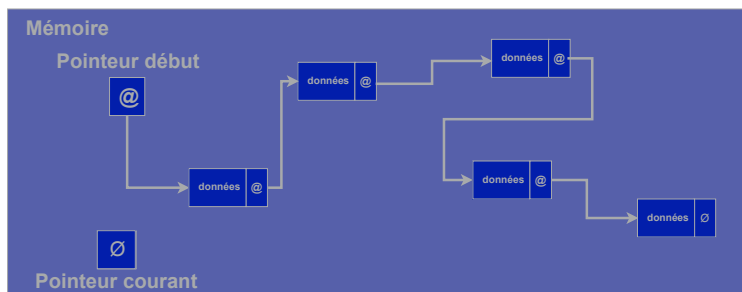
```
courant ← début // initialisation sur le début de la liste (si liste vide
// alors courant = NIL)
tant que courant ≠ NIL // on continue tant que l'on pointe sur un maillon
... courant; ... // traitement du maillon courant
courant ← courant;.svt // passage au maillon suivant
ftant
```



# Parcours de la liste

- Utilisation d'un pointeur de parcours :

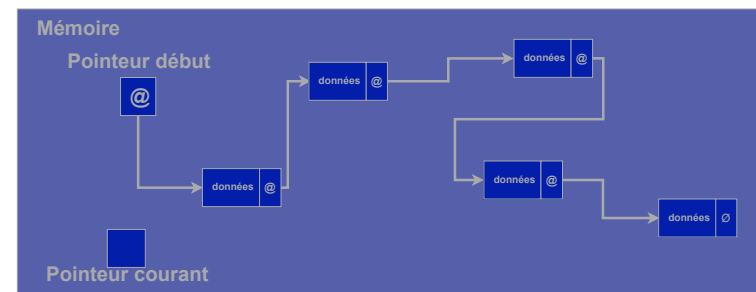
```
courant ← début // initialisation sur le début de la liste (si liste vide
// alors courant = NIL)
tant que courant ≠ NIL // on continue tant que l'on pointe sur un maillon
... courant; ... // traitement du maillon courant
courant ← courant;.svt // passage au maillon suivant
ftant
```



# Suppression d'un élément

- Au début de la liste :

```
courant ← début // on récupère le 1er maillon
début ← début;.svt // la liste commence au second maillon
// (l'ancien 1er n'est plus dedans)
Libère(courant) // on désalloue l'ancien 1er maillon
courant ← NIL // on réinitialise le pointeur courant pour éviter
// les accès non valides
```

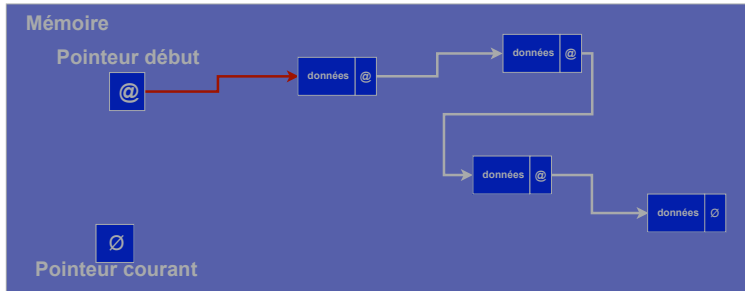


# Suppression d'un élément

- Au début de la liste :

```

courant ← début // on récupère le 1er maillon
début ← début↑.svt // la liste commence au second maillon
// (l'ancien 1er n'est plus dedans)
Libère(courant) // on désalloue l'ancien 1er maillon
courant ← NIL // on réinitialise le pointeur courant pour éviter
// les accès non valides
    
```

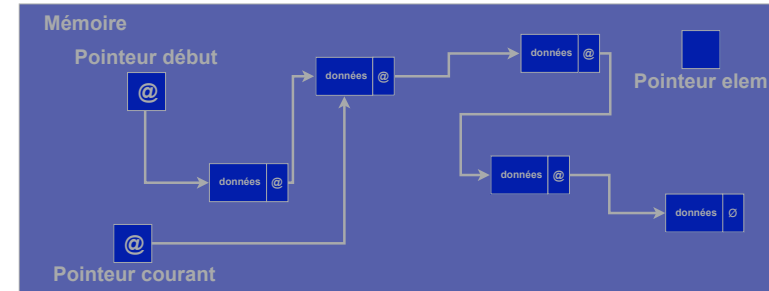


# Suppression d'un élément

- Dans la liste : il faut un pointeur sur le maillon précédent

```

elem ← courant↑.svt
si elem ≠ NIL alors
    courant↑.svt ← elem↑.svt
    Libère(elem)
    elem ← NIL
fsi
    
```

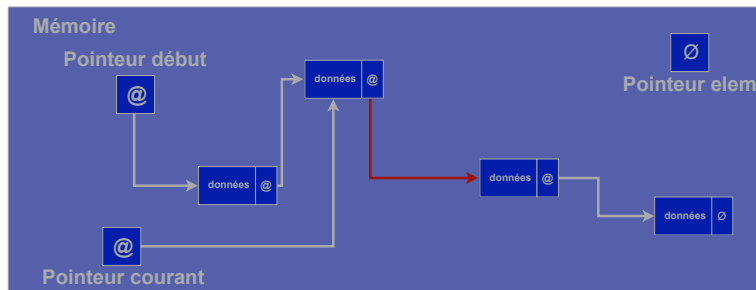


# Suppression d'un élément

- Dans la liste : il faut un pointeur sur le maillon précédent

```

elem ← courant↑.svt
si elem ≠ NIL alors
    courant↑.svt ← elem↑.svt
    Libère(elem)
    elem ← NIL
fsi
    
```

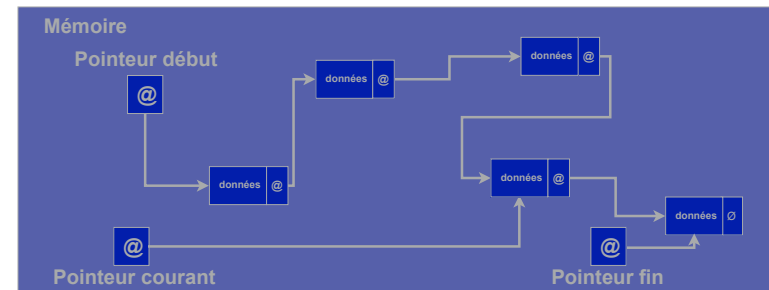


# Suppression d'un élément

- À la fin de la liste : idem avec les deux derniers maillons

```

Libère(fin) // on libère le dernier maillon
fin ← courant // l'avant-dernier maillon devient le dernier
fin↑.svt ← NIL // le suivant du dernier maillon est réinitialisé
// pour marquer la fin de la liste
    
```

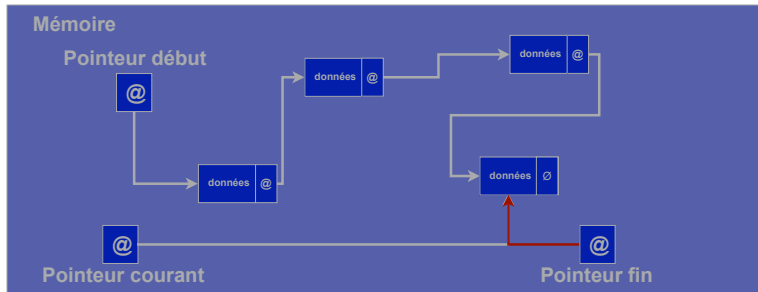


# Suppression d'un élément

- À la fin de la liste : idem avec les deux derniers maillons

```

Libère(fin) // on libère le dernier maillon
fin ← courant // l'avant-dernier maillon devient le dernier
fin↑.svt ← NIL // le suivant du dernier maillon est réinitialisé
// pour marquer la fin de la liste
    
```

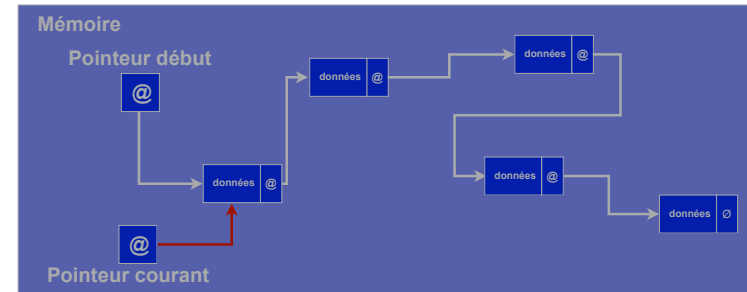


# Destruction de la liste

- Suite de suppressions du 1er élément de la liste :

```

courant ← début // initialisation sur le début de la liste
// (si liste vide alors courant = Nil)
tant que courant ≠ NIL // on continue tant que la liste n'est pas vide
    début ← courant↑.svt // la liste commence au second maillon
    Libère(courant) // suppression du 1er élément
    courant ← début // on se replace au début de la liste
ftant
    
```

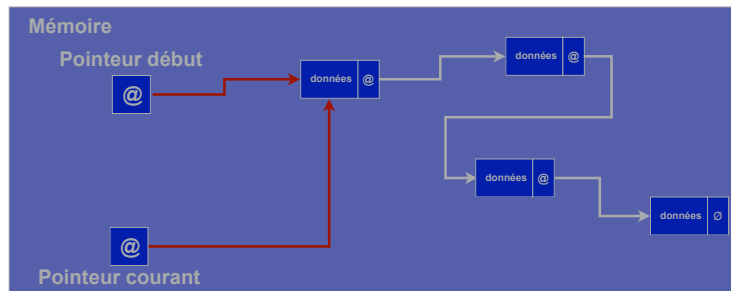


# Destruction de la liste

- Suite de suppressions du 1er élément de la liste :

```

courant ← début // initialisation sur le début de la liste
// (si liste vide alors courant = Nil)
tant que courant ≠ NIL // on continue tant que la liste n'est pas vide
    début ← courant↑.svt // la liste commence au second maillon
    Libère(courant) // suppression du 1er élément
    courant ← début // on se replace au début de la liste
ftant
    
```

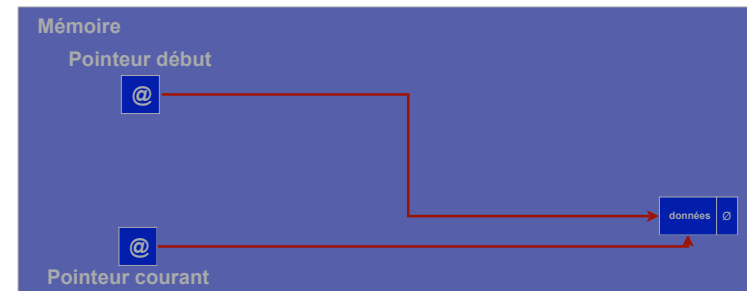


# Destruction de la liste

- Suite de suppressions du 1er élément de la liste :

```

courant ← début // initialisation sur le début de la liste
// (si liste vide alors courant = Nil)
tant que courant ≠ NIL // on continue tant que la liste n'est pas vide
    début ← courant↑.svt // la liste commence au second maillon
    Libère(courant) // suppression du 1er élément
    courant ← début // on se replace au début de la liste
ftant
    
```



# Destruction de la liste

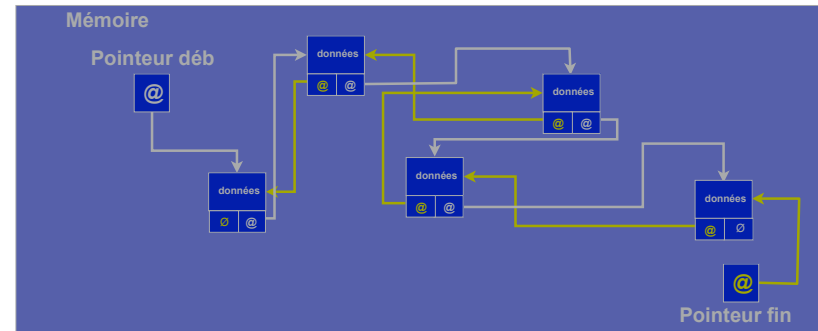
- Suite de suppressions du 1er élément de la liste :

```
courant ← début // initialisation sur le début de la liste
// (si liste vide alors courant = Nil)
tant que courant ≠ NIL // on continue tant que la liste n'est pas vide
  début ← courant↑.svt // la liste commence au second maillon
  Libère(courant) // suppression du 1er élément
  courant ← début // on se replace au début de la liste
ftant
```



# Listes doublement chaînées

- Présence d'un pointeur sur le maillon précédent :
  - ☺ Permet les parcours en sens inverse (un début et une fin)
  - ☺ Simplifie certaines manipulations (suppressions...)
  - ☹ Plus de liens à gérer entre les maillons



# Les fichiers

- Permettent un stockage permanent d'informations :
    - En entrée : liste de paramètres et/ou de données
    - En sortie : résultat(s) d'un algorithme stocké(s) dans un fichier
    - Intermédiaire : lorsque la mémoire vive n'est pas suffisante
  - Comme la mémoire vive, les *mémoires de masse permanentes* sont constituées d'un ensemble fini et homogène de *blocs élémentaires* d'information (*octets*)
  - ↳ Un fichier est une *zone finie logiquement contiguë* d'une telle mémoire
- Il est donc généralement défini par :
- Sa position de départ sur le média de stockage
  - Sa taille, exprimée en blocs élémentaires ou en unités logiques

# Contenu

- Bien qu'un fichier soit une suite de blocs élémentaires *identiques*, il peut contenir des informations de *types différents*
  - Son contenu *sémantique* dépend de la façon dont on *interprète* la suite des blocs élémentaires, c'est-à-dire de la façon dont on les regroupe sémantiquement
  - Par exemple, un fichier de 16 octets peut contenir :
    - 16 caractères (1 octet chacun)
    - 4 entiers standards (4 octets chacun)
    - 2 réels en précision double (8 octets chacun)
- Mais aussi :
- 1 entier suivi de 4 caractères suivis d'un réel double précision
  - 2 caractères, un entier, un caractère, un réel double, un caractère
  - ...

# Classification

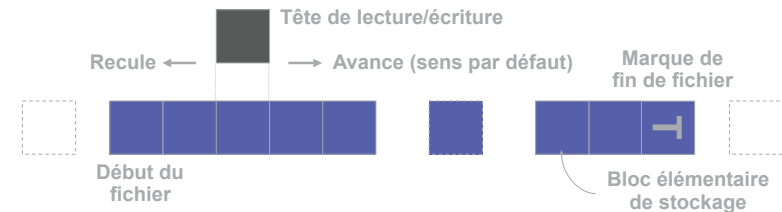
- Il existe plusieurs types d'organisation de fichier :
  - **Séquentiel** : (bandes magnétiques)
    - › Les éléments sont stockés les uns à la suite des autres dans l'ordre d'écriture et l'accès aux éléments se fait dans l'ordre de stockage
  - **Séquentiel indexé** : (idem avec marqueurs)
    - › Idem mais avec une table d'index ordonnée faisant une correspondance directe entre un index (clé) et l'élément correspondant dans le fichier
  - **Séquentiel à accès direct** : (disques durs)
    - › Organisation séquentielle par défaut mais avec la possibilité de se déplacer directement à une position donnée dans le fichier pour lire et/ou écrire
  - **Relatif** : (systèmes spécifiques)
    - › La position des éléments dans le fichier dépend d'un index particulier (ordonné) et non plus de l'ordre d'écriture (trous possibles dans le fichier)

## Opérations sur les fichiers

- **Ouverture** :
  - Selon 4 modes : lecture, écriture, ajout et modification
- **Accès aux éléments** :
  - Lecture ou écriture
- **Détection de la fin de fichier**
- **Fermeture**
- **Accès directs** :
  - Déplacement à une position donnée
  - Placement au début
  - Placement à la fin
  - Récupération de la position courante

# Fichiers séquentiels à accès direct

- Un fichier séquentiel à accès direct peut être vu comme une **suite linéaire** de blocs sur laquelle se déplace une **tête de lecture/écriture** :



- **Déclaration** :

fic	(fichier)	Un fichier	Le rôle dépend de l'utilisation du fichier
-----	-----------	------------	--

## Ouverture

- **Fonction d'ouverture** :  
fonction Ouvrir(in nom : chaîne de caractères, in mode : caractère) : ret fichier
  - **nom** contient le nom du fichier dans le système de stockage
  - **mode** indique le mode d'utilisation du fichier :
    - › 'L' : lecture → lectures uniquement
    - › 'E' : écriture → écritures uniquement
    - › 'A' : ajout → écritures à partir de la fin du fichier
    - › 'M' : modification → lecture/modification des éléments existants
- **Exemples** :

```
fic ← Ouvrir("liste.txt", 'L') // ouvre le fichier liste.txt
                               // en lecture
fic ← Ouvrir("toto", 'E')    // ouvre le fichier toto en écriture
fic ← Ouvrir("titi", 'A')    // ouvre le fichier titi en ajout
fic ← ouvrir("tutu", 'M')    // ouvre le fichier tutu en modification
```

# Modes d'ouverture

## Ouverture en lecture :

- La tête de L/E est placée *au début* du fichier
- ⚠ Aucun lecture n'est encore faite !



Ouverture d'un fichier non vide



Ouverture d'un fichier vide

## Ouverture en écriture :

- La tête de L/E est placée *au début* du fichier
- La marque de fin de fichier est placée au début
- ⚠ L'ouverture en écriture d'un fichier existant *vide* ce fichier !!
- Attention aux *destructions de données* !!!



Ouverture d'un fichier en écriture

# Modes d'ouverture (suite)

## Ouverture en ajout :

- La tête de L/E est placée *à la fin* du fichier
- Le contenu initial du fichier n'est pas modifié



Ouverture d'un fichier non vide



Ouverture d'un fichier vide

## Ouverture en modification :

- La tête de L/E est placée *au début* du fichier
- ⚠ Il faut respecter l'organisation sémantique du fichier !!



Ouverture d'un fichier non vide

# Accès aux éléments

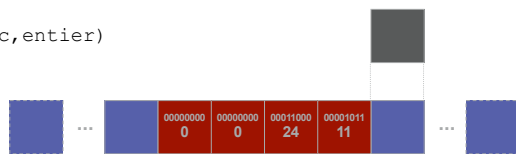
## Lecture :

fonction Lire(in fic: fichier, in typeElem : Type) : ret typeElem

- *fic* doit être un fichier ouvert en lecture ou modification
- *typeElem* indique le type de l'élément lu
- La fonction ne doit être appelée que lorsque l'on est sûr de ne pas être à la *fin de fichier* (test *avant* l'appel)
- Après la lecture, la tête de L/E se retrouve juste après l'élément lu

## Exemple :

val ← Lire(fic,entier)



val 123  
Valeur entière lue avec le codage **Big Endian**

# Accès aux éléments

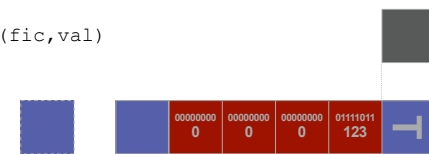
## Ecriture :

fonction Écrire(in fic: fichier, in val : typeElem) : ret booléen

- *fic* doit être un fichier ouvert en écriture, ajout ou modification
- *val* est l'élément à écrire dans le fichier
- La fonction retourne VRAI si l'écriture a réussi et FAUX sinon
- Après l'écriture, la tête de L/E se retrouve juste après l'élément écrit

## Exemple :

rés ← Écrire(fic,val)



val 123  
Octets écrits avec le codage **Big Endian**

## Fin de fichier et fermeture

- **Détection de la fin de fichier :**

fonction FinDeFic(in fic : fichier) : ret booléen

- *fic* doit être un fichier ouvert
- La fonction retourne VRAI si la tête de L/E se trouve en face de la marque de fin de fichier et FAUX sinon



- **Fermeture d'un fichier :**

fonction Fermer(in fic : fichier) : ret vide

- *fic* doit être un fichier ouvert
- Délimite la zone d'utilisation d'un fichier dans l'algorithme
- La fermeture est nécessaire pour changer de mode d'utilisation

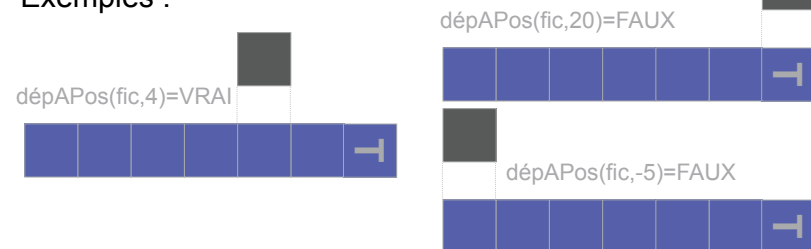
## Déplacements (1/2)

- Fonction de **déplacement quelconque** de la tête de L/E :

fonction DépAPos(in fic : fichier, in pos : entier) : ret booléen

- *fic* doit être un fichier ouvert
- *pos* est spécifiée par rapport au **début** du fichier (position 0)
- Si la position spécifiée n'est pas **dans** le fichier, la tête de L/E est placée sur l'extrémité correspondante et la fonction retourne FAUX, sinon elle retourne VRAI

- Exemples :



## Déplacements (2/2)

- Fonction de **placement** de la tête de L/E **au début** du fichier :

fonction DépAuDébut(in fic : fichier) : ret vide

- *fic* doit être un fichier ouvert sinon la fonction n'a pas d'effet
- Equivalent à DépAPos(fic,0)

- Fonction de **placement** de la tête de L/E **à la fin** du fichier :

fonction DépAFin(in fic : fichier) : ret vide

- *fic* doit être un fichier ouvert sinon la fonction n'a pas d'effet

- Fonction de **récupération** de la **position courante** :

fonction PosFic(in fic : fichier) : ret entier

- *fic* doit être un fichier ouvert sinon la fonction retourne -1
- Retourne la position de la tête de L/E par rapport au début du fichier

- Lorsque plusieurs fichiers sont ouverts simultanément, ces fonctions agissent **indépendamment** sur chaque fichier comme s'il y avait une tête de L/E par fichier

## Modes d'un fichier

- On distingue généralement **deux** modes de représentation des données dans un fichier :

- Le mode **binnaire** : équivalent à ce qui a été présenté précédemment
  - ⊕ Généralement beaucoup plus dense
  - ⊗ Le contenu du fichier n'est pas implicitement interprétable
  - ⊗ Problèmes de compatibilité (codages Big ou Little Endian)
- Le mode **texte** : le contenu du fichier est une suite de caractères
  - ⊕ Le contenu du fichier est implicitement interprétable le plus souvent
  - ⊕ Utilisable facilement sur tous les types de machines/systèmes
  - ⊗ Nécessite une conversion explicite des valeurs en leur équivalent textuel lors des écritures dans le fichier
  - ⊗ Nécessite la conversion inverse lors des lectures depuis le fichier
  - ⊗ Taille plus importante du fichier
- Les deux modes peuvent être mélangés : parties texte et parties binaires