

L'ALGORITHME

Écrire du pseudo-code



ALGORITHME : RECETTE DE CUISINE ?

Un algorithme est une suite logique d'instructions qui, quand elles sont exécutées correctement aboutissent au résultat attendu.

On peut les décrire de manière générale, identifier des procédures, des suites d'actions ou de manipulations précises à accomplir **séquentiellement**.

C'est cela, un algorithme : le concept qui traduit la notion intuitive de procédé systématique, applicable mécaniquement, sans réfléchir, en suivant un mode d'emploi précis.

Un algorithme c'est donc « presque » comme **une recette de cuisine**.



EXEMPLE

On fait tous de l'algorithme sans le savoir. Par exemple en cuisine, si on souhaite faire une omelette, on a des étapes à passer avant de pouvoir la manger...

Algorithme de l'omelette :

Prendre 6 œufs, un saladier, une fourchette et une poêle

Pour chaque nombre entre 1 et 6 (inclus tous les deux) :

- Casser un œuf dans le saladier

- Jeter la coquille à la poubelle

- Saler, poivrer et fouetter le contenu du saladier

- Faire cuire à la poêle

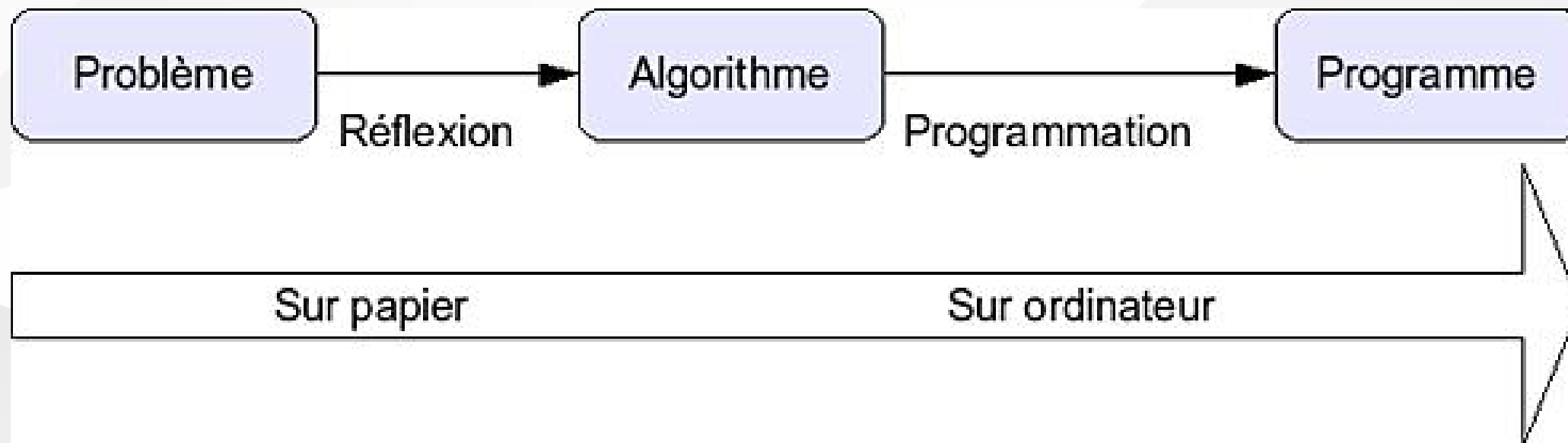
COMPÉTENCES POUR L'ALGORITHMIQUE !!

Être bon en mathématique ?

- **oui** et **non** !! Cela permet plus facilement d'arriver au résultat d'une problématique donnée.
- mais il faut simplement :
 - avoir de l'intuition : elle s'apprend par l'expérience
 - être logique : on ne fait pas d'omelettes sans casser les œufs au préalable.
 - être méthodique et rigoureux : toujours se mettre à la place de la machine!
- Enfin, la pratique vous permettra de gagner en efficacité.

Et La programmation ?

- Monsieur le formateur, c'est quand qu'on code ?... Pourquoi ne pas directement coder ?
1. **Attention : dans les tests techniques, c'est très souvent demandé d'écrire de l'algorithme !!!**
 2. Faire de l'algorithme permet d'apprendre sans les particularités d'un langage
 - En effet, moins de contraintes car c'est dans un langage naturel qu'on pratique tous les jours.



DE LA RÉFLEXION À LA PROGRAMMATION

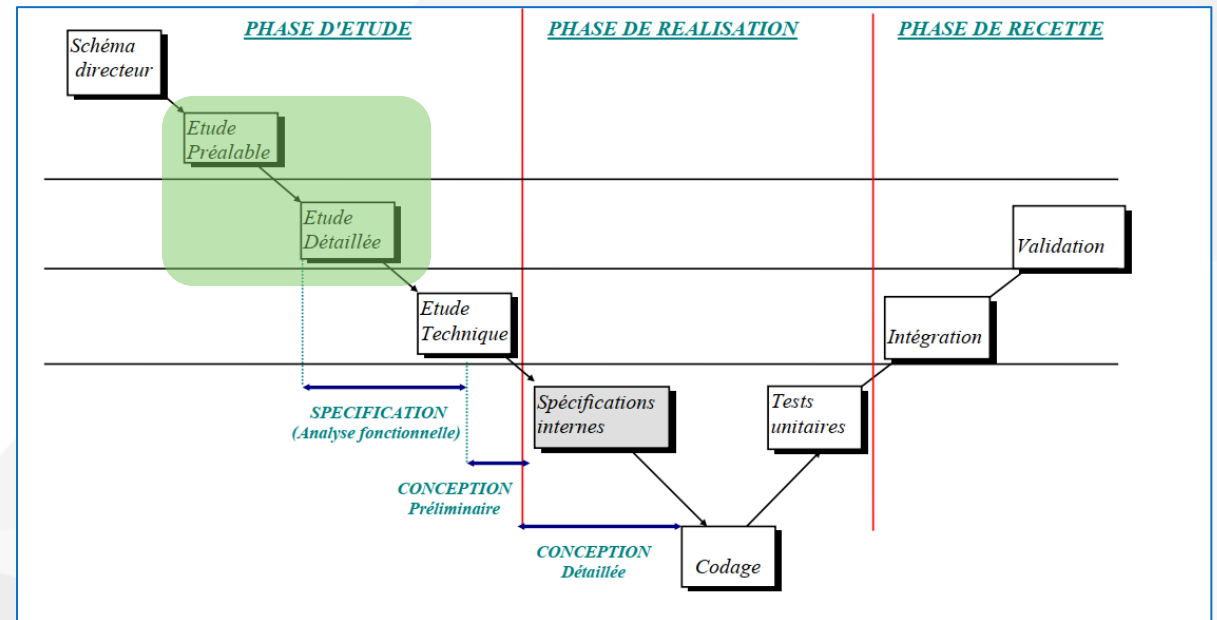


CYCLE DE VIE D'UN PROJET

Dans le cycle de vie d'un projet (ici le cycle en V), le processus d'écriture algorithmique vient dans la conception préliminaire - conception détaillée.

Même si avec l'expérience, l'écriture d'algorithmique peut sembler inutile, cela reste un bon moyen de mettre en place une documentation technique pour d'autres personnes et vous fera gagner du temps sur le codage en évitant les régressions du code.

Toujours penser à ceux qui vont relire même vous !!!



L'ALGORITHME TEXTUEL

On utilise le **pseudo-code** pour exprimer clairement et formellement un algorithme :

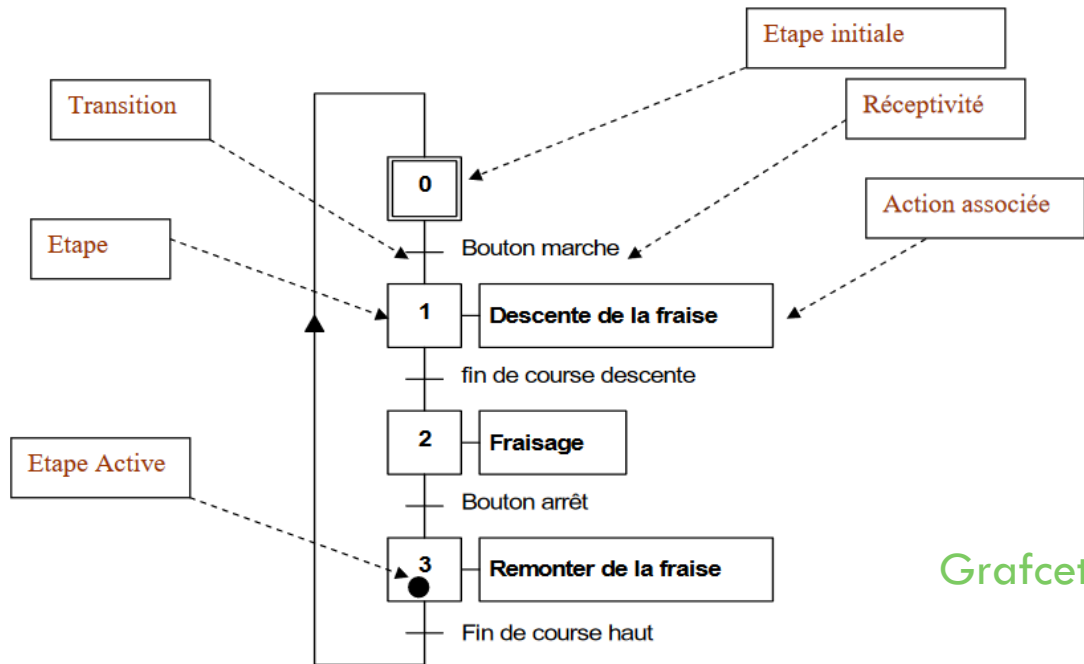
- Proche d'un langage de programmation **mais ce n'est pas du codage.**
- Expression d'idées formelles dans la **langue naturelle** de ses usagers en imposant une forme rigoureuse (**règles et normalisation**).
- Bien adapté aux problèmes informatiques et réponds bien à un type de problèmes donnés.
- Son formalisme a été repris dans de nombreux langages de programmation.

L'algorithme se définit suivant :

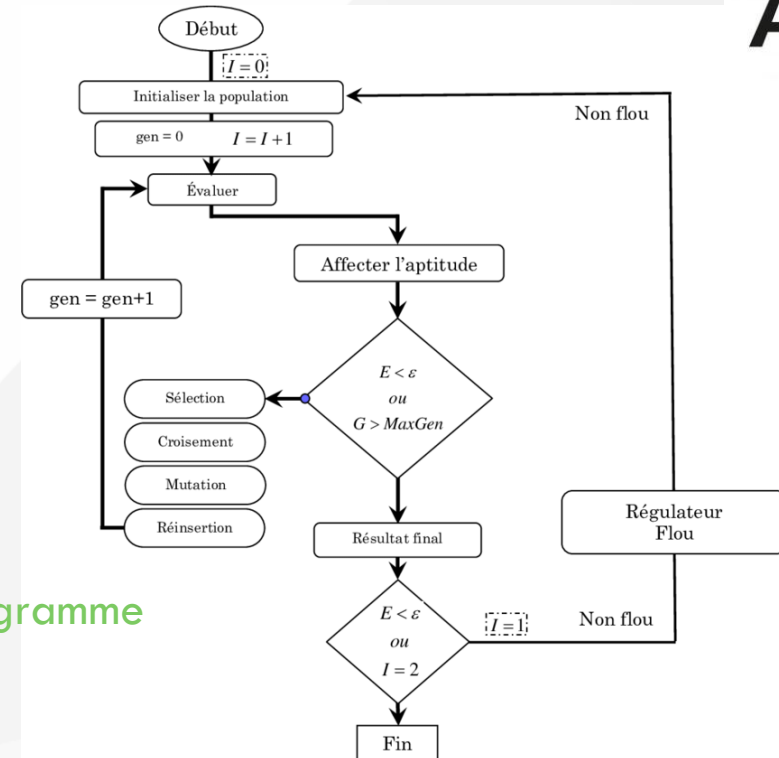
- La définition des "**entités**"
- L'utilisation des "entités" définies **les actions**
- La présentation
 - **Commentaires** et **l'indentation**
- L'esprit dans lesquels les algorithmes sont construits.

Autres formalismes :

- Graphiques : organigrammes, réseaux de Pétri, Grafcet.



Grafcet



Organigramme

REPRÉSENTATION EN GRAPHE



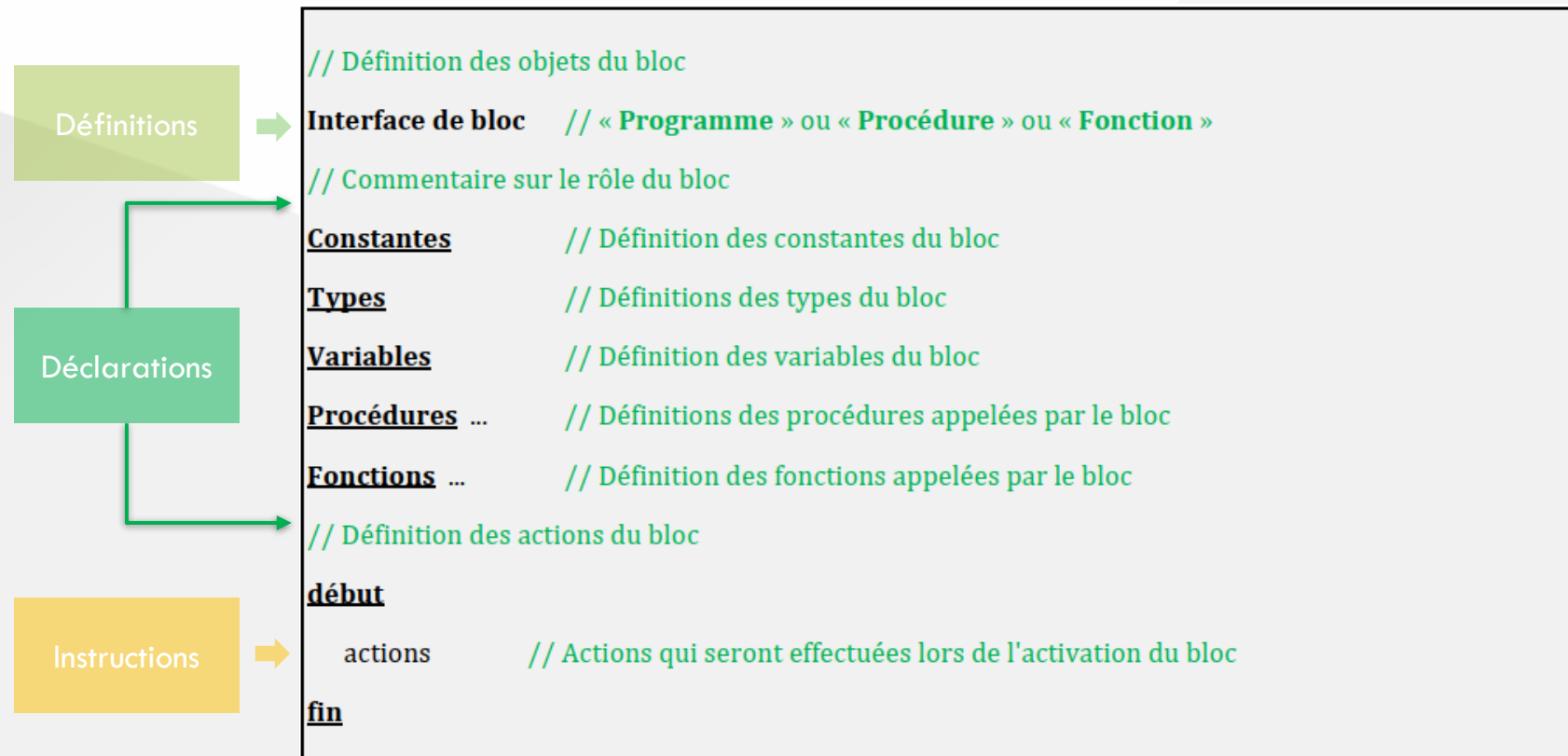


ÉCRIRE DU PSEUDO-CODE

DÉBUTONS PAR DE LA DÉFINITION

STRUCTURE D'UN ALGORITHME TEXTUEL

3 parties principales : Définitions / Déclarations / Instructions (actions)



NOTIONS D'ENTITÉS ET D'ACTIONS

Les entités forment l'ensemble des éléments qui sont manipulés dans un algorithme.

Plusieurs types d'entités :

- **Les types :**

- Défini par un indicateur ^{nom} et une référence à une famille (entier, réel etc..).
- Permet ainsi de classer les entités dans une famille, ainsi que les manipulations possibles sur ce type.

- **Les constantes :**

- Défini par un identificateur, représentée par une valeur et défini par un type. La constante est invariante au cours du temps. **Quel est l'intérêt d'une constante ? Faire en sorte que l'entité devienne invariante, empêchant toutes modifications.**

- **Les variables :**

- Défini par un **identificateur**, un **type** et un **contenu** qui va évoluer au cours du temps.

A QUOI SERVENT LES VARIABLES ?

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs.

- Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier), etc.
- Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs.
- Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, etc.


Toujours est-il que dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une variable.

En résumé :

- une variable a un type qui permet de définir la valeur qu'elle va contenir
- une variable est définie par un indicateur permettant de la nommer et de la retrouver facilement.
- une variable sera stockée dans la mémoire de l'ordinateur
- une variable pourra évoluer dans le temps, c'est-à-dire qu'on pourra lui **attribuer** plusieurs valeurs différentes. **Le contraire d'une variable est donc une constante.**

NOTIONS D'ENTITÉS ET D'ACTIONS

- Les procédures et fonctions.
 - Définie par un identificateur qui permet de l'appeler, les types des objets qu'elle va manipuler (**les paramètres**), les actions qu'elle effectue sur les objets et qui lui seront donnés à l'appel de la procédure.
 - Une **procédure exécute des instructions sans retourner de résultats à l'appelant.**
 - Une **fonction exécute des instructions retournant un résultat à l'appelant.**
- Les actions.
 - Opérations qui pourront être réalisées sur les entités définies dans l'algorithme.
 - Plusieurs types d'actions :
 - **Observation** : comparaison de deux objets.
 - **Modification** : donne une valeur à une variable.
 - **Alternative** : effectuer des actions suivant certaines conditions.
 - **Répétitive** : itérer des actions selon une condition de terminaison.
 - **Complexe** : appel d'une procédure ou une fonction.



LE LANGAGE ALGORITHMIQUE

ÉLÉMENTS DU LANGAGE

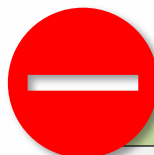
CARACTÈRES UTILISÉS ET MOTS RÉSERVÉS ET SYMBOLES



Les majuscules	A ... Z
Les minuscules	a ... z
Les chiffres	0 ... 9
Les signes	_ = < > ' () [] * + - / , : . Espace



Conventions de Nommage



alors	autrecas	booléen	caractère
choix	constantes	créer	de
début	détruire	div	écrire
enregistrement	entier	entrée	et
faire	faux	fermer	fichier
fin	finchoix	finenregistrement	finfichier
finsi	fintantque	fonction	indexé
jusqu'à	lire	mod	non
null	ou	ouvrir	pointeur
positionner	procédure	programme	quelconque
réel	répéter	retourner	si
sinon	sortie	sur	tableau
tantque	types	variables	vrai
:=	<>	>=	<=
//	->		

RÈGLES ET CONVENTIONS

Règles

- doit être **le plus lisible possible**, de manière que n'importe qui d'autre que l'auteur soit capable de comprendre en le lisant.
- ne doit pas être trop long (une page écran).
 - S'il est trop long, il faut le découper en fonctions et procédures.
- Les structures de contrôle doivent être indentées. Il en est de même pour les répétitives.
 - A chaque imbrication d'une structure de contrôle, on décale d'une tabulation.

Conventions

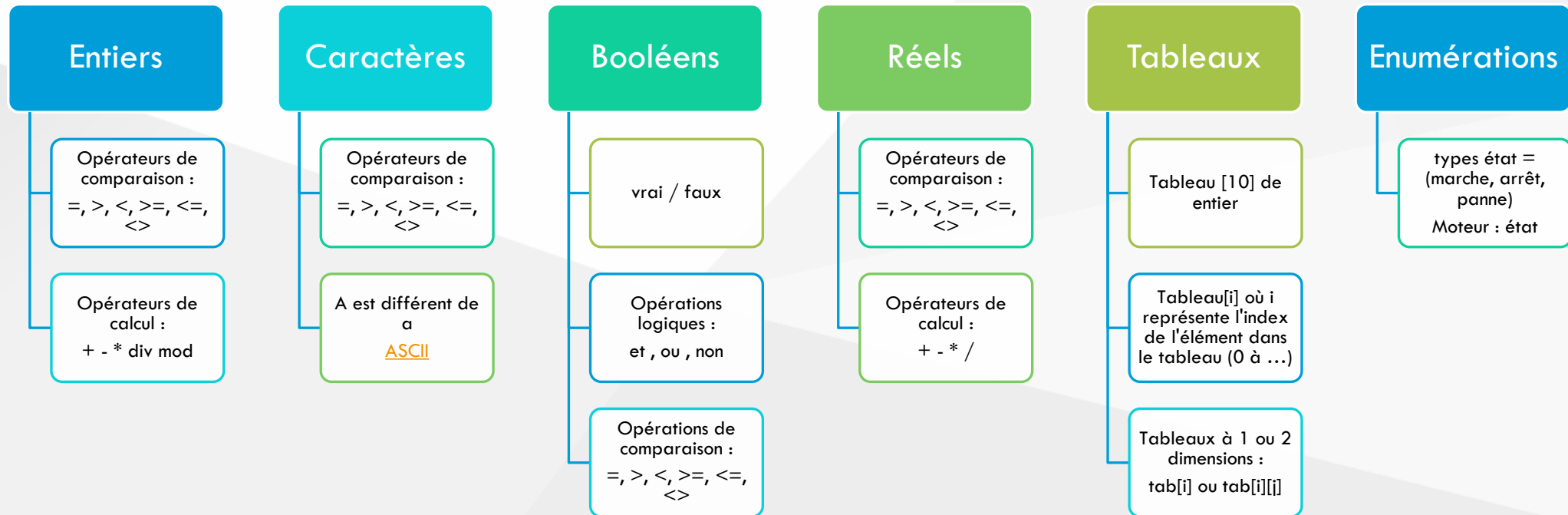
- Le nom des variables doit être significatif, c'est à dire indiquer clairement à quoi elles servent.
- On doit toujours être capable de donner un nom significatif à une procédure ou à une fonction.
- Le nombre de paramètres ne doit être trop grand (en général inférieur à 5) car cela nuit à la lisibilité du programme.
- Une procédure ou une fonction doit être la plus générale possible de manière à pouvoir être réutilisée dans d'autres circonstances.
- Si le but d'une procédure est de calculer une valeur simple, il est préférable d'en faire une fonction.
- Il est souvent plus clair d'écrire une fonction booléenne plutôt qu'une condition complexe.

LES FAMILLES

Pour simplifier le traitement des chaînes de caractères, on peut partir du principe d'un type Chaîne

texte : Chaîne // déclaration

Texte[0] correspondra au 1^{er} caractère de la chaîne



LES INSTRUCTIONS

- Déclaration du programme principal

```
Programme nomProgramme
... // section des déclarations constantes, variables, types, fonctions, procédures
Début
|   ...
Fin
```

- Déclaration de constantes

Constante NOM : [Type] = valeur

Constante PI : Réel = 3,141559

- Déclaration de variables

Variable nom : [Type] = valeur

Variable compteurLettres : Entier

VISIBILITÉ DES OBJETS.

Le principe est simple : un objet est visible (utilisable) dans l'unité algorithmique qui l'a défini, et dans toutes les unités algorithmiques définies dans celle-ci.

Mais il est interdit d'utiliser des variables dans une procédure ou fonction, qui ne seraient pas définies dans la procédure ou fonction ou en paramètre de celle-ci.

Notions de variables globales et locales :

- Une variable est dite **locale** si elle est définie dans l'environnement d'où on la regarde.
- Elle est dite **globale** si elle est définie dans un environnement qui est le père, ou un aïeul, de cet environnement.
- Donc une même variable peut être considérée globale ou locale relativement à l'endroit d'où elle est vue.

LES INSTRUCTIONS

- Déclaration de fonctions

```
Fonction nomFonction(param1 : [Type], param2 : [Type]...) : [TypeRésultat]
Constante ...
Variables
    resultat : TypeRésultat
Début
    ...
    // renvoi le résultat à l'appelant
    retour resultat
Fin
```

- Appel de fonction

variable ← nomFonction(param1, param2...)

résultat ← racine(69) ou Afficher(Racine(69))

Affectation ou assignation

Le symbole ← représente une affectation qui consiste d'attribuer une valeur à une variable.

On peut aussi utiliser := voir = mais peut être confondu avec un test

LES INSTRUCTIONS

- Déclaration de procédures

```
Procédure nomProcédure(param1 : [Type], param2 : [Type]...)  
|  
|  Constante ...  
|  Variable ...  
|  Début  
|  |  Actions  
|  Fin
```

- Appel de procédure

nomProcédure(param1, param2...)

afficher(Bonjour)



LES ENTRÉES ET SORTIES

INTERACTIONS UTILISATEUR AVEC LE PROGRAMME

ENTRÉE / SORTIE

- À la console

Il existe des actions complexes qui permettent :

- Lire des informations au clavier.
 - Lire(nombre) où nombre est une variable qu'on aura définie préalablement.
 - *Affectera le résultat de la lecture dans la variable*
- Ecrire des informations à l'écran.
 - Ecrire('le résultat est : ', nombre)

• Fichier

Nous pouvons ouvrir un fichier, lire un élément du fichier, écrire un élément dans le fichier, fermer le fichier, et tester la fin du fichier :

- Déclarer un fichier
 - nomfic : fichier de type élément
- Ouvrir un fichier
 - ouvrir('fichier.txt', nomfic)
- Lire un fichier
 - lire(nomfic, variable)
- Écrire dans un fichier
 - écrire(nomfic, valeur)
- fermer un fichier
 - fermer(nomfic)
- tester un fichier
 - Si finfichier(nomfic) alors ...

QUELQUES EXEMPLES

Algorithme par la pratique.
apprendre à lire et à écrire nos premiers algorithmes



PARTIE 1

Exercice 1.1

Quelles seront les valeurs des variables A et B après exécutions des instructions suivantes ?

Variable A, B : entier

Début

A := 1

B := A + 3

A := 3

Fin

Exercice 1.2

Quelles seront les valeurs des variables A, B et C après exécutions des instructions suivantes ?

Variable A, B, C : entier

Début

A := 5

B := 3

C := A + B

A := 2

C := B - A

Fin

PARTIE 1

Exercice 1.3

Quelles seront les valeurs des variables A et B après exécutions des instructions suivantes ?

Variable A, B : entier

Début

A := 5

B := A + 4

A := A + 1

B := A - 4

Fin

Exercice 1.4

Quelles seront les valeurs des variables A, B et C après exécutions des instructions suivantes ?

Variable A, B, C : entier

Début

A := 3

B := 10

C := A + B

B := A + B

A := C

Fin

PARTIE 1

Exercice 1.5

Que produit l'algorithme suivant ?

Variable A, B, C : caractères

Début

A := "423"

B := "12"

C := A + B

Fin



PARTIE 1

A
réaliser

Exercice 1.6

C'est un classique, écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable (Exemple : des entiers) ?

Exercice 1.7

Variante du 1.6, on dispose de trois variables A, B et C.

Ecrivez un algorithme transférant à B la valeur de A, à C la valeur de B et à A la valeur de C (toujours quels que soient les contenus préalables de ces variables) (Exemple : des entiers) ?

QUELQUES EXEMPLES

Algorithme par la pratique.
apprendre à écrire nos premiers algorithmes
Utilisation de fonctions et de procédures



PARTIE 2

A
réaliser

Exercice 2.1

Ecrire un algorithme qui demande à l'utilisateur le prix Hors taxe d'un objet et qui donne sa valeur TTC (multiplier le prix par 1.196).

Exercice 2.2

Ecrire un algorithme qui demande à l'utilisateur son prénom et son nom et qui affiche ensuite la phrase "Bonjour *prénom* votre nom est *nom*"

Variante : avec une fonction et une procédure



LES OPÉRATEURS

AGIR AVEC NOS ENTITÉS

LES OPÉRATEURS

Nature	Variables utilisées	Notation	Signification
Opérateurs arithmétiques	Entier Réal	+	Addition
		-	Soustraction
		*	Multiplication
		/	Division (réelle)
		DIV	Division entière
		MOD	Reste de la division entière
Opérateurs logiques	Booléen Entier	et	Fonction ET
		ou	Fonction OU
		ouex	Fonction OU EXCLUSIF
		non	Fonction NON
Opérateur de concaténation	Chaîne de caractères	+	Concaténation
Opérateurs de comparaison	Booléen Entier Réal Caractère Chaîne de caractères	=	Egal
		≠	Différent
		<	Inférieur
		>	Supérieur
		≤	Inférieur ou égal
		≥	Supérieur ou égal



LES TESTS ET CONDITIONS

LES CONDITIONS

Une condition est une comparaison

Elle signifie qu'une condition est composée de trois éléments :

- une valeur
- un opérateur de comparaison
- une autre valeur.

Les valeurs peuvent être a priori de n'importe quel type (numériques, caractères...).

- Mais si l'on veut que la comparaison ait un sens, il faut que les deux valeurs de la comparaison soient du même type

Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-contre.

Prenons le cas Toto est inclus entre 5 et 8.

- En fait cette phrase cache non une, mais deux conditions. Car elle revient à dire que Toto est supérieur à 5 et Toto est inférieur à 8.
- Il y a donc bien là deux conditions, reliées par ce qu'on appelle un opérateur logique, le mot ET.

L'informatique met à notre disposition quatre opérateurs logiques : ET, OU, NON, et XOR.

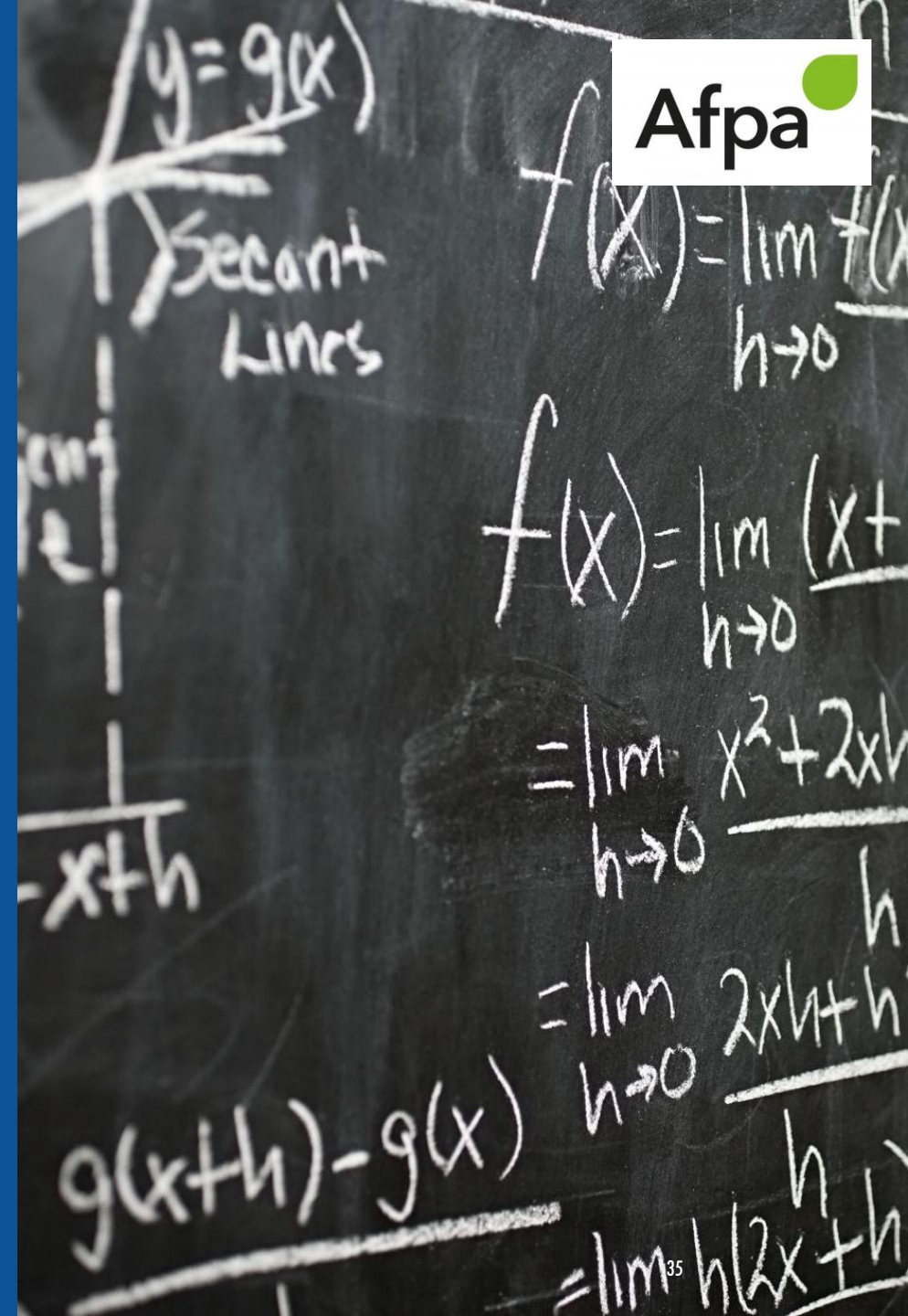
ALGÈBRE DE BOOLE

L'algèbre de Boole permet d'effectuer des opérations entre des entrées et des sorties, et utilise des portes logiques pour faire modifier les signaux et avoir des sorties potentiellement différentes des entrées.

Elle utilise principalement les portes logiques :





- ET : Les deux entrées doivent être vérifiées pour que la sortie se déclenche.
- OU : Au moins une des deux entrées doit être vérifiée pour que la sortie se déclenche.
- NON : La sortie est l'inverse de l'entrée.
- OU EXCLUSIF : Moins souvent utilisée en programmation classique (mais plutôt en électronique), c'est une combinaison de ET, de OU et de NON.

Une fois le schéma logique fait, on peut facilement passer au code sans trop de difficultés.



LES TABLES DE VÉRITÉ

Symboles des portes logiques

NOT	
AND	
OR	
XOR	

ET (AND)		
$A \& B$		
A	B	$A.B$
0	0	0
0	1	0
1	0	0
1	1	1

OU (OR)		
$A B$		
A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	1

XOU (XOR)		
$A \oplus B$		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

NON (NOT)	
$!A$	
A	\bar{A}
1	0
0	1

MISE EN PLACE DES TESTS

Les tests vont servir à conditionner nos actions.

- **Si** une condition du test est respectée **alors** l'action peut se dérouler.

Il existe plusieurs types de structures de contrôle permettant de mettre en place tout un ensemble d'enchainements d'actions conditionnés avec les tests.

- **Si** une condition du test est respectée **alors** l'action peut se dérouler **Sinon** j'effectue une autre action.

C'est à cela que vont nous servir les tables de vérité car on va pouvoir composer des conditions.

Quelques exemples :

Si Toute valeur supérieure à 18 alors...

Valeur > 18

Si Toute valeur inférieur à 18 ou supérieur à 25 alors...

Valeur < 18 ou valeur > 25

Si Le courant est présent et le bouton Power est sur marche alors...

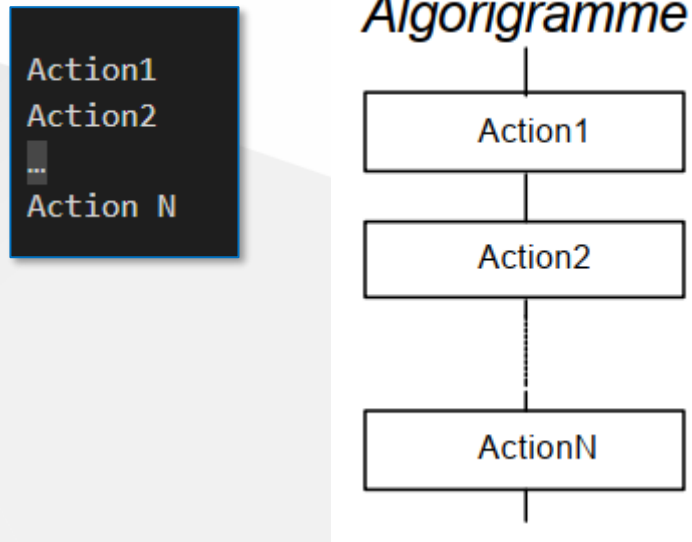
Courant = alimenté et Power = marche



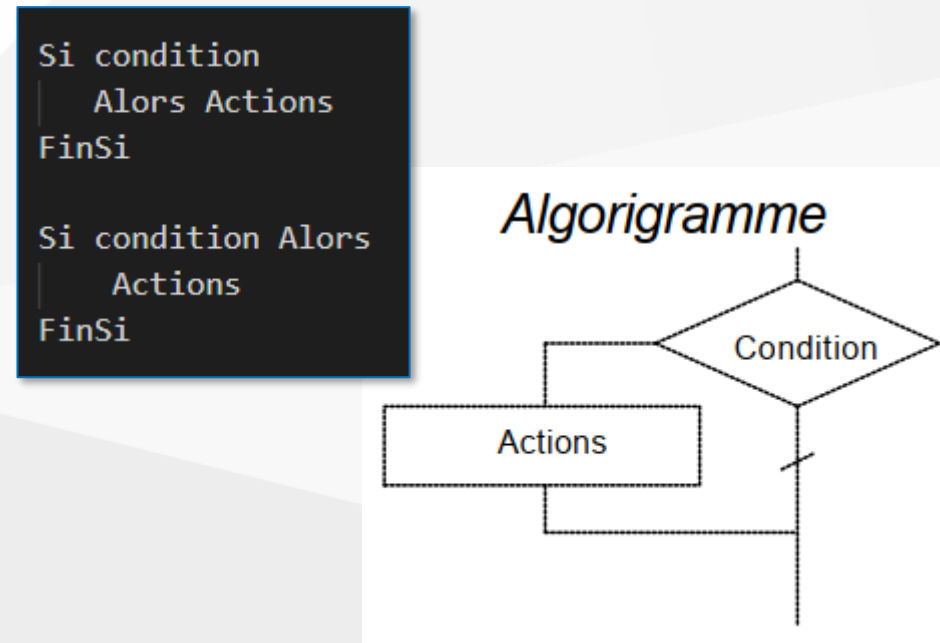
LES STRUCTURES ALGORITHMIQUES

LES STRUCTURES ALGORITHMIQUES

Séquence linéaire :



Structure Si ... Alors ... :

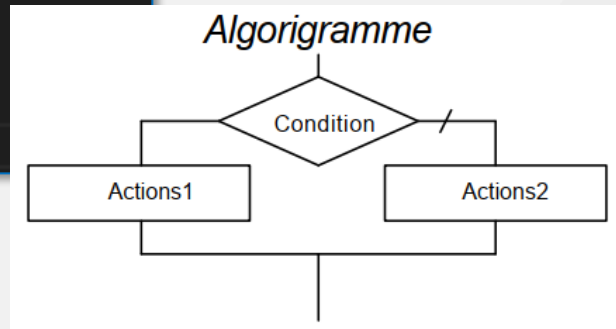


LES STRUCTURES ALGORITHMIQUES

Structure Si ... Alors ... Sinon ... :

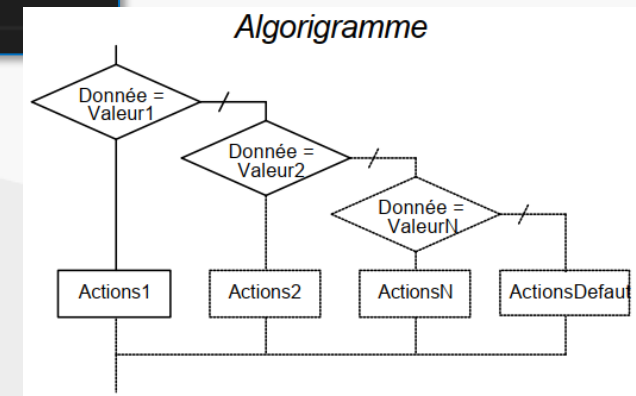
```
Si condition
  Alors Action1
  Sinon Action2
FinSi

Si condition Alors
  Action1
Sinon
  Action2
FinSi
```



Structure Choix multiple :

```
Cas où donnée Vaut
  Valeur1 : Actions1
  Valeur2 : Actions2
  ...
  ValeurN : ActionsN
  Autre : ActionsDéfaut
FinCas
```

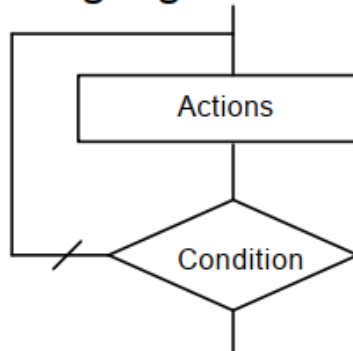


LES STRUCTURES ALGORITHMIQUES

Structure Répéter ... Jusqu'à :

```
Répéter  
| Actions  
Jusqu'à Condition
```

Algorithme



La vérification de la condition s'effectue après les actions. Celles-ci sont donc exécutées au moins un fois.

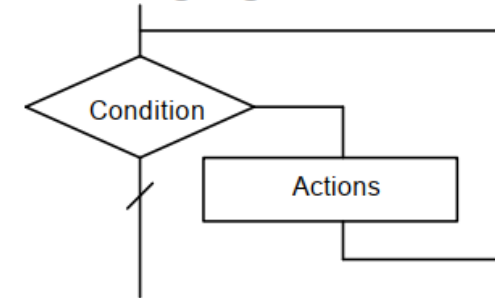
Structure Tant que ... Faire ...

```
TantQue Condition  
| Faire Actions  
FinTantQue
```



La vérification de la condition s'effectue avant les actions. Celles-ci peuvent donc ne jamais être exécutées.

Algorithme



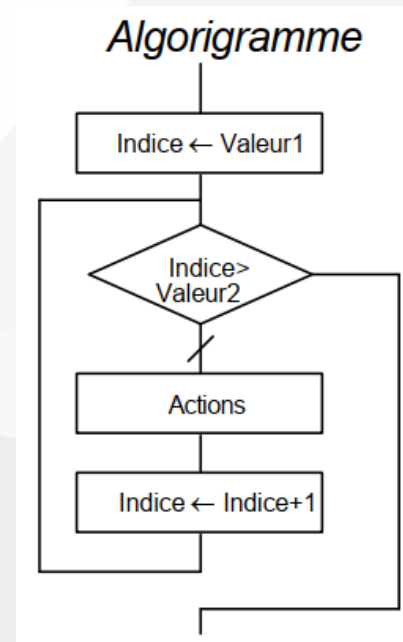
LES STRUCTURES ALGORITHMIQUES

Structure Pour indice allant de ... a ...
faire :

```
Pour indice Allant de valeur1 à valeur2  
  Faire  
    Actions...  
    indice := indice + 1  
  FinFaire  
FinPour
```



Dans ce type de
boucle, on inclue un
pas d'incrmentation
(+1, +2 ...)





LES TABLEAUX

MANIPULATION TABLEAU, LISTE...

Pour rappel, la création d'un tableau :

```
Variable tab : Tableau[10] d'Entier
```

- tab sera un tableau qui va contenir 10 entiers.
- Le premier élément de notre tableau se trouvera à l'indice 1
- Le dernier élément de notre tableau se trouvera à l'indice 10



Nous sommes en pseudo-code, d'où volontairement je commence à 1 comme indice du 1^{er} élément de mon tableau.

Accès à un élément de notre tableau :

- Tab[indice] où indice correspondra à l'emplacement de l'élément ciblé.

Parcours de notre tableau :

- Pour parcourir l'ensemble de notre tableau, nous allons utiliser la structure Pour...

```
Pour indice Allant de 1 à longueur_tableau
Faire
    // ex : si on souhaite écrire dans le tableau
    Lire tableau[indice]

    // ex : si on souhaite lire le contenu du tableau
    Ecrire tableau[indice]
FinFaire
FinPour
```

QUELQUES EXEMPLES

Algorithme par la pratique.

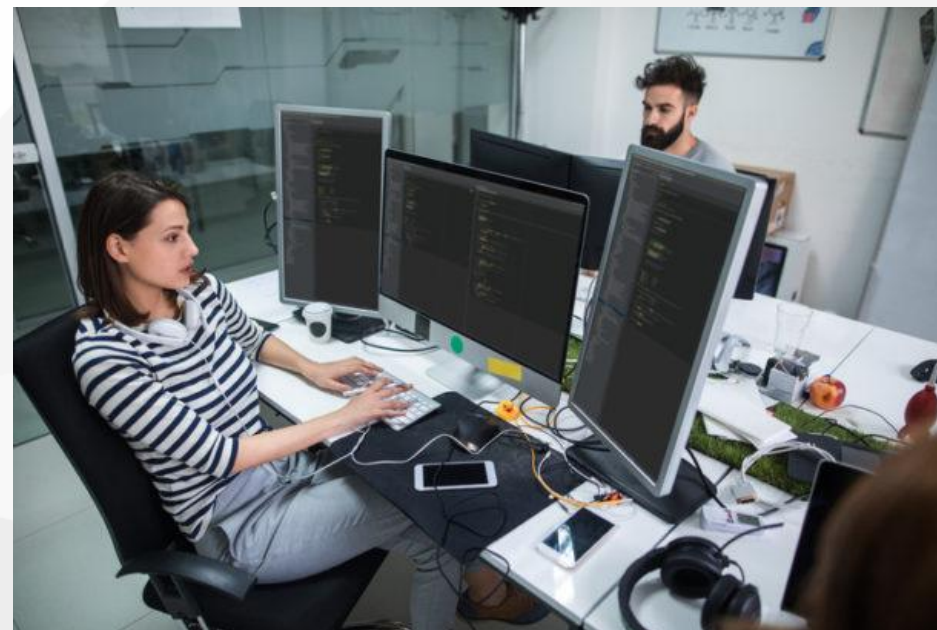


PARTIE 2

A
réaliser

Exercice 2.3

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif, positif, ou nul et afficher le résultat du produit obtenu ?



PARTIE 3

A
réaliser

Exercice 3.1

Ecrire un algorithme qui demande un nombre compris entre 10 et 20, jusqu'à ce que la réponse convienne. En cas de réponse supérieure à 20, on fera apparaître un message : "plus petit" et inversement "plus grand" si le nombre est inférieur à 10.

Exercice 3.2

Ecrire un algorithme qui demande un nombre de départ et qui calcule sa factorielle.

Note : la factorielle de 8 vaut : $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$

EXEMPLE

Énoncé :

Écrire un algorithme qui demande à l'utilisateur d'entrer un nombre entier positif et affiche tous les nombres pairs de 0 jusqu'à ce nombre.

Instructions :

- Demandez à l'utilisateur d'entrer un nombre entier positif et stockez-le dans une variable.
- Vérifiez si le nombre est positif. Si ce n'est pas le cas, affichez un message d'erreur et demandez à l'utilisateur de saisir à nouveau un nombre positif.
- À l'aide d'une boucle, parcourez tous les nombres de 0 jusqu'au nombre saisi par l'utilisateur.
- Pour chaque nombre, vérifiez s'il est pair. Si c'est le cas, affichez-le.
- Répétez les étapes 1 à 4 jusqu'à ce que tous les nombres pairs de 0 jusqu'au nombre saisi aient été affichés.

```
Algorithme afficherNombresPairs()  
    Début  
        Variable n : Entier  
        Écrire "Entrez un nombre entier positif : "  
        Lire n  
        Tant que n <= 0 Faire  
            Écrire "Le nombre doit être positif. Réessayez : "  
            Lire n  
        Fin Tant que  
        Écrire "Les nombres pairs jusqu'à ", n, " sont :"  
        Pour i de 0 à n Faire  
            Si i mod 2 = 0 Alors  
                Écrire i  
            Fin Si  
        Fin Pour  
    Fin
```


LA RECETTE DE LA SENSEO



```
// Programme SENSEO
// Auteur : Jérôme BOEBION
// Description : Algo simple de la machine senseo
Programme SENSEO
Types
    Etat = (fixe, clignotant)
Constantes
    TASSE : Entier = 10 ml
    TEMPERATURE : Entier = 89 // température max de l'eau
Variables
    active : Booleen // vrai indique si la machine est en fonctionnement
    choix_1_cafe, choix_2_cafe : Booleen // choix 1 ou 2 café
    voyant : Etat // indique l'état du voyant fixe pour prête, clignotant pour en attente
    temperatureEau : Entier // temperature de l'eau
    niveauEau : Entier // niveau de l'eau

Procédures
Procédure chaufferEau()
Début
    TantQue temperatureEau < TEMPERATURE Faire
        activer la resistance de la senseo
    FinTantQue
    voyant = fixe
Fin

Procédure ajouterEau()
Début
    TantQue niveauEau < TASSE Faire
        attendre que l'eau soit remise
    FinTantQue
    voyant = fixe
Fin
```

LA RECETTE DE LA SENSEO (SUITE)



```

Procédure faireCafe(compteur : Entier)
// BLOC DE DECLARATION DE LA PROCEDURE

Début
    TantQue niveauEau > TASSE Faire
        Cas où compteur Vaut
            1 : niveauEau = niveauEau - TASSE
            2 : niveauEau = niveauEau - (2*TASSE)
        FinCas
        Si (niveauEau < TASSE OU temperatureEau < TEMPERATURE) Alors
            voyant = clignotant
        FinSi
    FinTantQue
Fin

// DEBUT DU PROGRAMME
Début
    TantQue active Faire

        // test de la temperature
        Si (temperatureEau < TEMPERATURE ET niveauEau > TASSE) Alors
            voyant = clignotant
            chaufferEau()
        FinSi
        Si

            // test du niveau de l'Eau
            Si niveauEau < TASSE Alors
                voyant = clignotant
                ajouterEau()
            FinSi

            // Tout est ok pour faire le café
            Si (voyant = fixe) Alors
                Si choix_1_cafe Alors
                    faireCafe(1)
                FinSi
                Si choix_2_cafe Alors
                    faireCafe(2)
                FinSi
            FinSi
        FinTantQue
    Fin

```

LES POMMES DE TERRE

Eplucher x pommes de terre en allant les chercher à la cave avec un panier ne pouvant contenir que 3 pommes de terre à la fois.

Ecrire l'algorithme permettant de réaliser ces actions ? Combien d'aller-retour est nécessaire pour éplucher toutes les pommes de terre ? Afficher le nombre d'aller-retours



Pseudo-code

```
// Epluche PommeDeTerre
Programme EpluchePommeDeTerre
// déclaration
Variable tour, totalPdt, pdt : Entier
Constante PANIER : Entier vaut 3
```

Début

```
// initialisation
tour := 0
```

```
Ecrire "Nombre total de pommes de terre ?"
Lire totalPdt
```

```
// sauvegarde quantité
pdt := totalPdt
```

```
// calcul des aller-retours
Tant Que (pdt >= 3) Faire
    pdt := pdt - PANIER
    tour := tour + 1
Fin Tant que
```

```
// test pour connaitre s'il reste encore moins de 3 Pommes de terre
Si (pdt > 0) Alors
    tour := tour + 1
Fin si
```

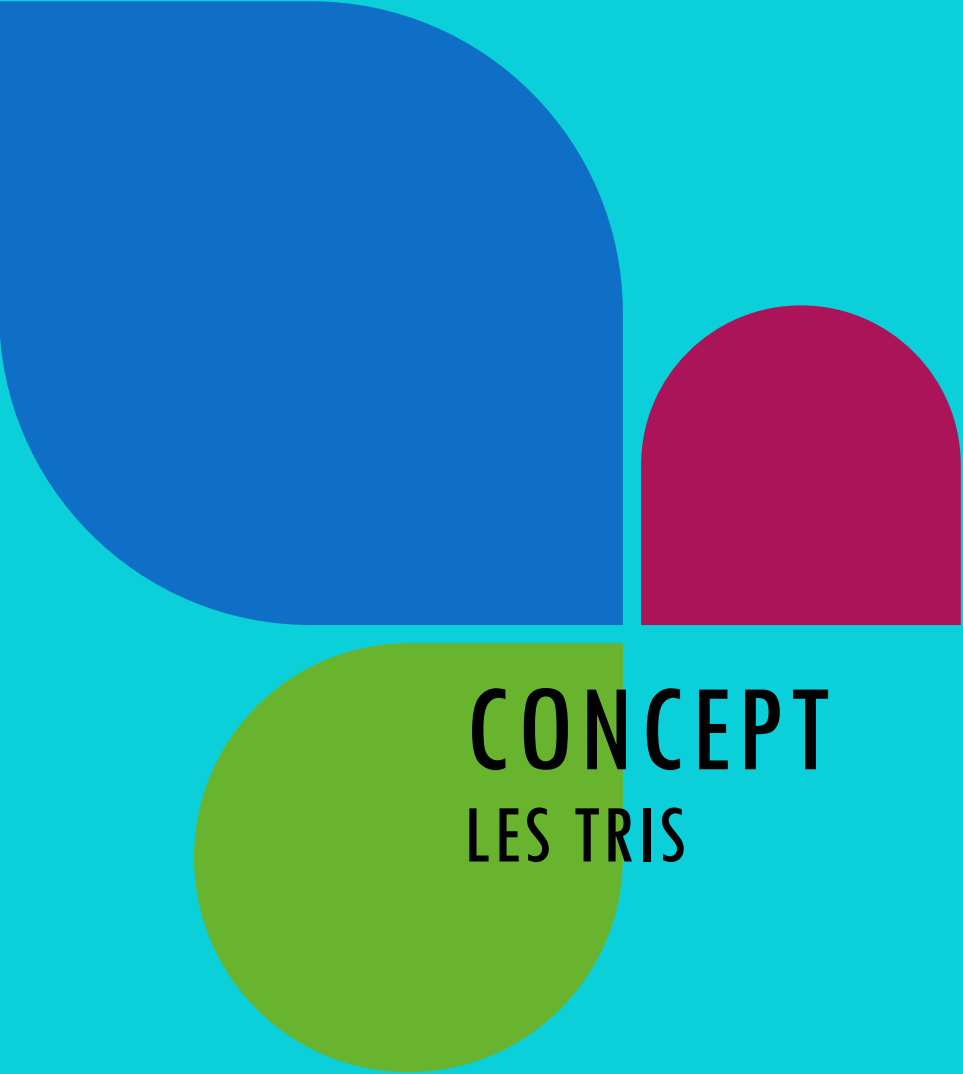
```
Ecrire "Pour éplucher ", totalPdt, " il faut ", tour, " allers-retours."
```

Fin

AlgoBox

```
1 FONCTIONS_UTILISEES
2
3 VARIABLES
4 tour EST_DU_TYPE NOMBRE
5 totalPdt EST_DU_TYPE NOMBRE
6 pommeDeTerre EST_DU_TYPE NOMBRE
7 panier EST_DU_TYPE NOMBRE
8 DEBUT_ALGORITHME
9 // initialisation des VARIABLES
10 panier PREND_LA_VALEUR 3
11 tour PREND_LA_VALEUR 0
12
13 // Affichage message
14 AFFICHER "Nombre total de pommes de terre ?"
15 LIRE totalPdt
16
17 // sauvegarde
18 pommeDeTerre PREND_LA_VALEUR totalPdt
19
20 // contrôle tant que total différent de la fin
21 TANT_QUE (pommeDeTerre >= 3) FAIRE
22     DEBUT_TANT_QUE
23         pommeDeTerre PREND_LA_VALEUR pommeDeTerre - panier
24         tour PREND_LA_VALEUR tour + 1
25     FIN_TANT_QUE
26
27 // test pour vérifier la présence ou pas d'un panier inférieur à 3
28 SI (pommeDeTerre > 0) ALORS
29     DEBUT_SI
30         tour PREND_LA_VALEUR tour + 1
31     FIN_SI
32
33 AFFICHER "Pour éplucher "
34 AFFICHER totalPdt
35 AFFICHER ", il faut "
36 AFFICHER tour
37 AFFICHER* " allers-retours"
38 FIN_ALGORITHME
```

VECTOR SKETCH



CONCEPT

LES TRIS

LES TRIS

Il existe de nombreux algorithmes de tri, chacun avec ses avantages et ses limites en termes de complexité temporelle, de stabilité, d'utilisation de mémoire, etc.

Voici quelques-uns des principaux algorithmes de tri

https://fr.wikipedia.org/wiki/Algorithme_de_tri

http://lwh.free.fr/pages/algo/tri/comparaison_tri.html

Dans les tests techniques en entreprise, c'est souvent demandé de trier un tableau en utilisant ces algorithmes

1. **Tri à bulle (Bubble Sort)** : compare les éléments adjacents et les échange si nécessaire. Il répète ce processus jusqu'à ce que la liste soit entièrement triée.
2. **Tri par sélection (Selection Sort)** : recherche à chaque étape le plus petit élément non trié dans la liste et l'échange avec l'élément à la position actuelle. Il répète ce processus pour tous les éléments jusqu'à ce que la liste soit triée.
3. **Tri par insertion (Insertion Sort)** : trie la liste en insérant chaque élément à sa place dans la portion déjà triée de la liste. À chaque étape, il sélectionne un élément non trié, le compare avec les éléments précédents et le déplace vers la bonne position.
4. **Tri par fusion (Merge Sort)** : basé sur le principe de diviser pour régner. Il divise la liste en deux parties égales, trie chaque partie de manière récursive, puis fusionne les parties triées pour obtenir une liste globalement triée.
5. **Tri rapide (Quick Sort)** : Également basé sur le principe de diviser pour régner, sélectionne un élément appelé pivot, réarrange la liste de manière à placer tous les éléments plus petits que le pivot avant lui, et tous les éléments plus grands après lui. Il répète ensuite ce processus récursivement pour les sous-listes avant et après le pivot.

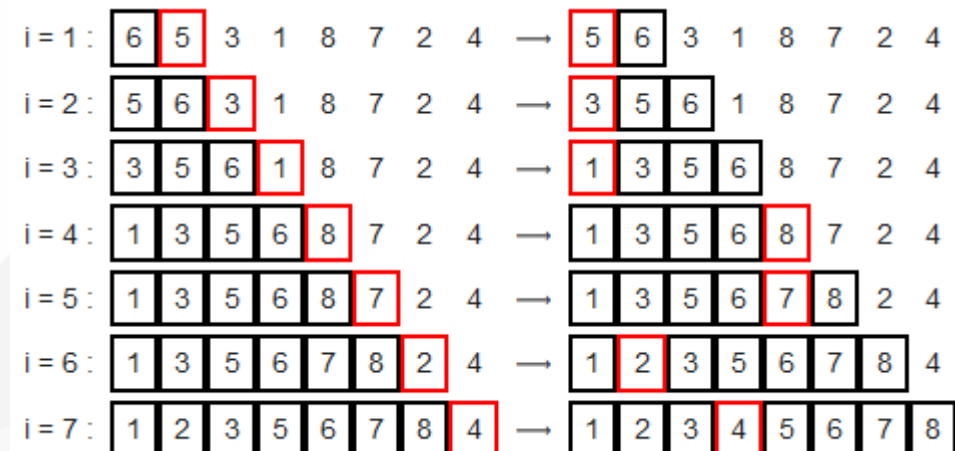
TRI PAR INSERTION

Tri par insertion (assez lent avec beaucoup de données mais efficace sur peu) : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis on met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier.

On parcourt le tableau en passant sur chacun des éléments à trier et on l'insère directement à la place où il devrait être. Pour faire cette insertion, on va simplement comparer l'élément courant avec chaque élément déjà trié sur la gauche.

- Efficace sur des entrées de petite taille.
- Efficace lorsque les données sont déjà presque triées. utilisé en pratique en combinaison avec d'autres méthodes comme le tri rapide.

<https://www.youtube.com/watch?v=bRPHvWgc6YM&list=WL&index=67>



TRI PAR INSERTION

Voici l'algorithme du tri par insertion :

On commence par rechercher, parmi les valeurs, quel est le plus petit élément, et où il se trouve. On l'identifie et on l'échange alors avec le premier élément.

On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus).

Et cetera, et cetera, jusqu'à l'avant dernier

```
procédure tri_insertion(tableau)
  pour i de 1 à longueur(tableau) - 1 faire
    clé = tableau[i]
    j = i - 1
    tant que j >= 0 et tableau[j] > clé faire
      tableau[j + 1] = tableau[j]
      j = j - 1
    fin tant que
    tableau[j + 1] = clé
  fin pour
fin procédure
```


TRI PAR INSERTION

Voici la version testée dans AlgoBox

Résultats

```
***Algorithme lancé***  
Tableau non trié  
4251333  
1233345  
***Algorithme terminé***
```

```
FONCTIONS_UTILISEES  
FONCTION afficherTableau(tab)  
  VARIABLES_FONCTION  
    i EST_DU_TYPE NOMBRE  
  DEBUT_FONCTION  
    POUR i ALLANT_DE 0 A tableau.length-1  
      DEBUT_POUR  
        AFFICHER tableau[i]  
      FIN_POUR  
    AFFICHER " "  
  FIN_FONCTION  
FONCTION initialisationTableau( )  
  VARIABLES_FONCTION  
  DEBUT_FONCTION  
    // initialisation du tableau avec des valeurs  
    tableau[0] PREND_LA_VALEUR 4:2:5:1:3:3:3  
  FIN_FONCTION  
VARIABLES  
  tableau EST_DU_TYPE LISTE  
  indice EST_DU_TYPE NOMBRE  
  cle EST_DU_TYPE NOMBRE  
  j EST_DU_TYPE NOMBRE  
DEBUT_ALGORITHME  
  // initialisation du tableau avec des valeurs  
  APPELER_FONCTION initialisationTableau()  
  // affichage du tableau initial  
  AFFICHER "Tableau non trié"  
  APPELER_FONCTION afficherTableau(tableau)  
  // trie du tableau  
  POUR indice ALLANT_DE 1 A tableau.length-1  
    DEBUT_POUR  
      cle PREND_LA_VALEUR tableau[indice]  
      j PREND_LA_VALEUR indice-1  
      TANT_QUE (j >= 0 ET tableau[j] > cle) FAIRE  
        DEBUT_TANT_QUE  
          tableau[j+1] PREND_LA_VALEUR tableau[j]  
          j PREND_LA_VALEUR j - 1  
        FIN_TANT_QUE  
      tableau[j + 1] PREND_LA_VALEUR cle  
    FIN_POUR  
  APPELER_FONCTION afficherTableau(tableau)  
FIN_ALGORITHME
```

LA RECHERCHE DANS UN TABLEAU

Nous allons maintenant nous intéresser au maniement habile d'une variable booléenne : la technique dite du **flag**.

Le **flag**, en anglais, est un petit drapeau, qui va rester baissé aussi longtemps que l'événement attendu ne se produit pas.

Et, aussitôt que cet événement a lieu, le petit drapeau se lève (la variable booléenne change de valeur).

Ainsi, la valeur finale de la variable booléenne permet au programmeur de savoir si l'événement a eu lieu ou non.

Soit un tableau comportant des valeurs. On doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.

- L'utilisation du flag va nous permettre d'indiquer dans le parcours du tableau si l'élément recherché est présent ou pas.
- Ensuite après avoir parcouru le tableau, nous allons pouvoir tester la valeur du flag et ainsi selon sa valeur, indiquer à l'utilisateur la présence ou non de la valeur recherchée dans le tableau.

```
Ecrire "Entrez la valeur à rechercher"
Lire N
flag ← Faux
Pour i ← 0 à taille
    Si N = Tab(i) Alors
        flag ← Vrai
    FinSi
FinPour
Si Trouvé Alors
    Ecrire "N fait partie du tableau"
Sinon
    Ecrire "N ne fait pas partie du tableau"
FinSi
```

TRI À BULLES = tri du tableau + flag

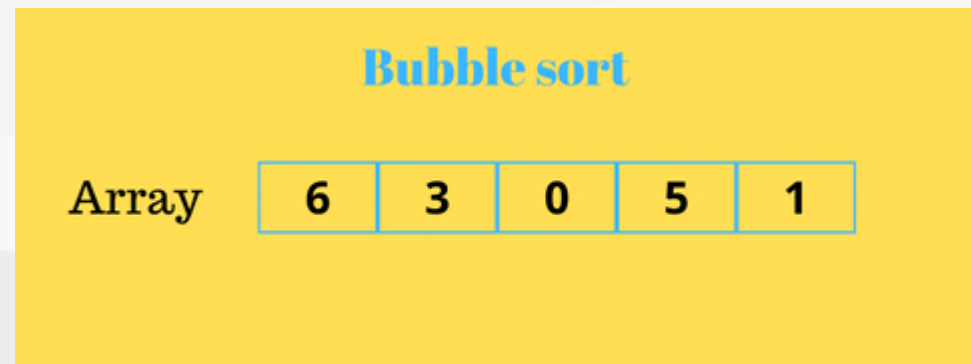
Le tri à bulles est de loin le plus simple de tous les algos de tri.

- *D'ailleurs c'est pour ça qu'il est super lent et quasiment jamais utilisé.*
- *Algorithme de tri simple mais peu efficace pour de grandes quantités de données.*
- *Pour des performances optimales, d'autres algorithmes de tri tels que le tri fusion ou le tri rapide sont préférables.*

La logique est simple :

- On passe sur chaque élément du tableau et on le compare à son voisin de droite.
- Si le voisin de droite est plus petit alors les deux éléments permutent, car l'élément le plus petit devrait être à gauche.
- On fait autant de passe que nécessaire jusqu'à ce que tout le tableau soit trié.

La condition d'arrêt étant le fait qu'aucune permutation (flag) n'ait été nécessaire dans une passe.



TRI À BULLES



Sur la deuxième boucle, il est à noter qu'on sait que l'élément le plus grand est positionné à la fin à chaque boucle. C'est pourquoi, on réduit de i à chaque nouvelle boucle.

Application du tri à bulles au tableau de nombres « 5 1 4 2 8 » ; pour chaque étape, les éléments comparés sont en gras.

Étape 1.

- 1.1. (**5** 1 4 2 8) → (**1** **5** 4 2 8). Les nombres 5 et 1 sont comparés, et comme $5 > 1$, l'algorithme les permute.
- 1.2. (1 **5** 4 2 8) → (1 **4** **5** 2 8). Permutation, car $5 > 4$.
- 1.3. (1 4 **5** 2 8) → (1 4 **2** **5** 8). Permutation, car $5 > 2$.
- 1.4. (1 4 2 **5** 8) → (1 4 2 **5** 8). Pas de permutation, car $5 < 8$.

À la fin de cette étape, un nombre est à sa place définitive, le plus grand : 8.

Étape 2.

- 2.1. (**1** **4** 2 5 8) → (**1** **4** 2 5 8). Pas de permutation.
- 2.2. (1 **4** **2** 5 8) → (1 **2** **4** 5 8). Permutation.
- 2.3. (1 2 **4** **5** 8) → (1 2 **4** **5** 8). Pas de permutation.

5 et 8 ne sont pas comparés puisqu'on sait que le 8 est déjà à sa place définitive.

Par hasard, tous les nombres sont déjà triés, mais cela n'est pas encore détecté par l'algorithme.

Étape 3.

- 3.1. (**1** **2** 4 5 8) → (**1** **2** 4 5 8). Pas de permutation.
- 3.2. (1 **2** **4** 5 8) → (1 **2** **4** 5 8). Pas de permutation.

Les deux derniers nombres sont exclus des comparaisons, puisqu'on sait qu'ils sont déjà à leur place définitive.

Puisqu'il n'y a eu aucune permutation durant cette étape 3, le tri optimisé se termine.

Étape 4.

- 4.1. (**1** **2** 4 5 8) → (**1** **2** 4 5 8). Pas de permutation.

Le tri est terminé, car on sait que les 4 plus grands nombres, et donc aussi le 5^e, sont à leur place définitive.

```
// Boucle principale : nombre de passages dans le tableau
POUR i DE 0 À TAILLE(tableau) - 2 FAIRE

    // À chaque passage, on compare les paires voisines
    POUR j DE 0 À TAILLE(tableau) - 2 - i FAIRE

        // Si deux éléments sont dans le mauvais ordre, on les échange
        SI tableau[j] > tableau[j + 1] ALORS
            temp ← tableau[j]
            tableau[j] ← tableau[j + 1]
            tableau[j + 1] ← temp
        FIN_SI
    FIN_POUR
FIN_POUR
```

TRI À BULLES

Voici la version testée dans AlgoBox

Résultats

```
***Algorithme lancé***  
2 - 5 - 8 - 6 - 1 - 3 - 1  
2  
3  
5  
6  
8  
***Algorithme terminé***
```

```
FONCTIONS_UTILISEES  
VARIABLES  
    tableau EST_DU_TYPE LISTE  
    i EST_DU_TYPE NOMBRE  
    j EST_DU_TYPE NOMBRE  
    temp EST_DU_TYPE NOMBRE  
DEBUT_ALGORITHME  
    tableau[0] PREND_LA_VALEUR 2:5:8:6:1:3  
    //// tableau non trié  
    POUR i ALLANT_DE 0 A tableau.length-1  
        DEBUT_POUR  
            AFFICHER tableau[i]  
            AFFICHER " - "  
        FIN_POUR  
    POUR i ALLANT_DE 0 A tableau.length-2  
        DEBUT_POUR  
            POUR j ALLANT_DE 0 A tableau.length-2-i  
                DEBUT_POUR  
                    SI (tableau[j] > tableau[j+1]) ALORS  
                        DEBUT_SI  
                            temp PREND_LA_VALEUR tableau[j]  
                            tableau[j] PREND_LA_VALEUR tableau[j+1]  
                            tableau[j+1] PREND_LA_VALEUR temp  
                        FIN_SI  
                    FIN_POUR  
                FIN_POUR  
            FIN_POUR  
    //// affichage tableau trié  
    POUR i ALLANT_DE 0 A tableau.length-1  
        DEBUT_POUR  
            AFFICHER tableau[i]  
        FIN_POUR
```

TRI PAR SÉLECTION

Tri par sélection : Algorithme de tri par comparaison

Sur un tableau de n éléments (numérotés de 0 à $n-1$), le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

• <https://www.youtube.com/watch?v=8u3Yq-5DTN8&list=WL&index=68>

Principe et fonctionnement de tri par sélection

List no Triée

comparer
élément

Sélect élément

List triée

7	12	17	29	33	25	40	42
7	12	17	25	33	29	40	42
7	12	17	25	33	29	40	42
7	12	17	25	33	29	40	42
7	12	17	25	29	33	40	42
7	12	17	25	29	33	40	42

TRI PAR SÉLECTION

Voici l'algorithme du tri par sélection

```
procédure tri_selection(tableau)
  pour i de 0 à longueur(tableau) - 2
    posmini ← i
    pour j de i + 1 à longueur(tableau) - 1
      si tableau(j) < tableau(posmini) alors
        posmini ← j
    fin si
  fin pour
  // échange
  temp ← tableau(posmini)
  tableau(posmini) ← tableau(i)
  tableau(i) ← temp
fin pour
fin procédure
```

TRI PAR SELECTION

Voici la version testée dans AlgoBox

Résultats

```
***Algorithme lancé***
0286290
2
2
6
8
9
***Algorithme terminé***
```

Code de l'algorithme

```
1  FONCTIONS_UTILISEES
2  VARIABLES
3    i EST_DU_TYPE NOMBRE
4    j EST_DU_TYPE NOMBRE
5    posmini EST_DU_TYPE NOMBRE
6    temp EST_DU_TYPE NOMBRE
7    tableau EST_DU_TYPE LISTE
8  DEBUT_ALGORITHME
9    tableau[0] PREND_LA_VALEUR 0:2:8:6:2:9
10   POUR i ALLANT_DE 0 A tableau.length-1
11     DEBUT_POUR
12       AFFICHER tableau[i]
13     FIN_POUR
14   POUR i ALLANT_DE 0 A tableau.length-2
15     DEBUT_POUR
16       posmini PREND_LA_VALEUR i
17       POUR j ALLANT_DE i A tableau.length-1
18         DEBUT_POUR
19           SI (tableau[j] < tableau[posmini]) ALORS
20             DEBUT_SI
21               posmini PREND_LA_VALEUR j
22             FIN_SI
23           FIN_POUR
24         //// echange
25         temp PREND_LA_VALEUR tableau[posmini]
26         tableau[posmini] PREND_LA_VALEUR tableau[i]
27         tableau[i] PREND_LA_VALEUR temp
28       FIN_POUR
29     //affiche tableau trié
30   POUR i ALLANT_DE 0 A tableau.length-1
31     DEBUT_POUR
32       AFFICHER tableau[i]
33     FIN_POUR
34
35
36
37  FIN_ALGORITHME
```


LE TRI PIVOT (TRI RAPIDE) QUICKSORT

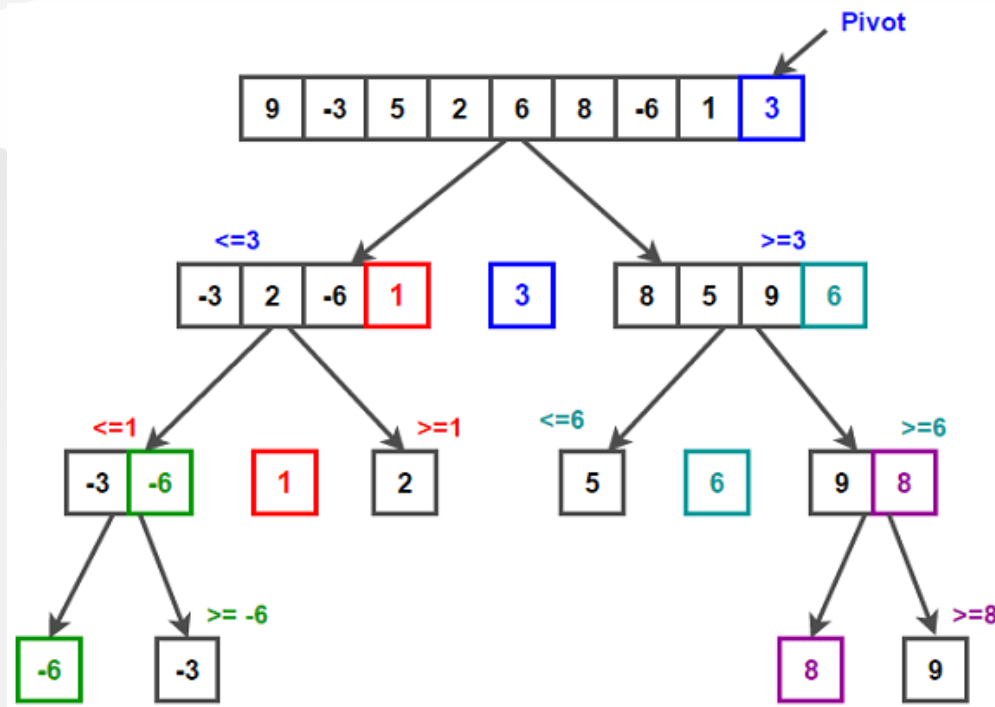
La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.

Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Concrètement, pour partitionner un sous-tableau :

- le pivot est placé à la fin (arbitrairement), en l'échangeant avec le dernier élément du sous-tableau
- tous les éléments inférieurs au pivot sont placés en début du sous-tableau
- le pivot est déplacé à la fin des éléments déplacés.

LE TRI PIVOT (TRI RAPIDE) QUICKSORT



```
partitionner(tableau T, entier premier, entier dernier, entier pivot)
// échange le pivot avec le dernier du tableau
    échanger T[pivot] et T[dernier]

    j := premier
    pour i de premier à dernier - 1
        // la boucle se termine quand i = dernier du tableau
        si T[i] <= T[dernier] alors
            échanger T[i] et T[j]
            j := j + 1
    échanger T[dernier] et T[j]
    renvoyer j

tri_rapide(tableau T, entier premier, entier dernier)
    si premier < dernier alors
        pivot := choix_pivot(T, premier, dernier)
        pivot := partitionner(T, premier, dernier, pivot)
        tri_rapide(T, premier, pivot-1)
        tri_rapide(T, pivot+1, dernier)
```

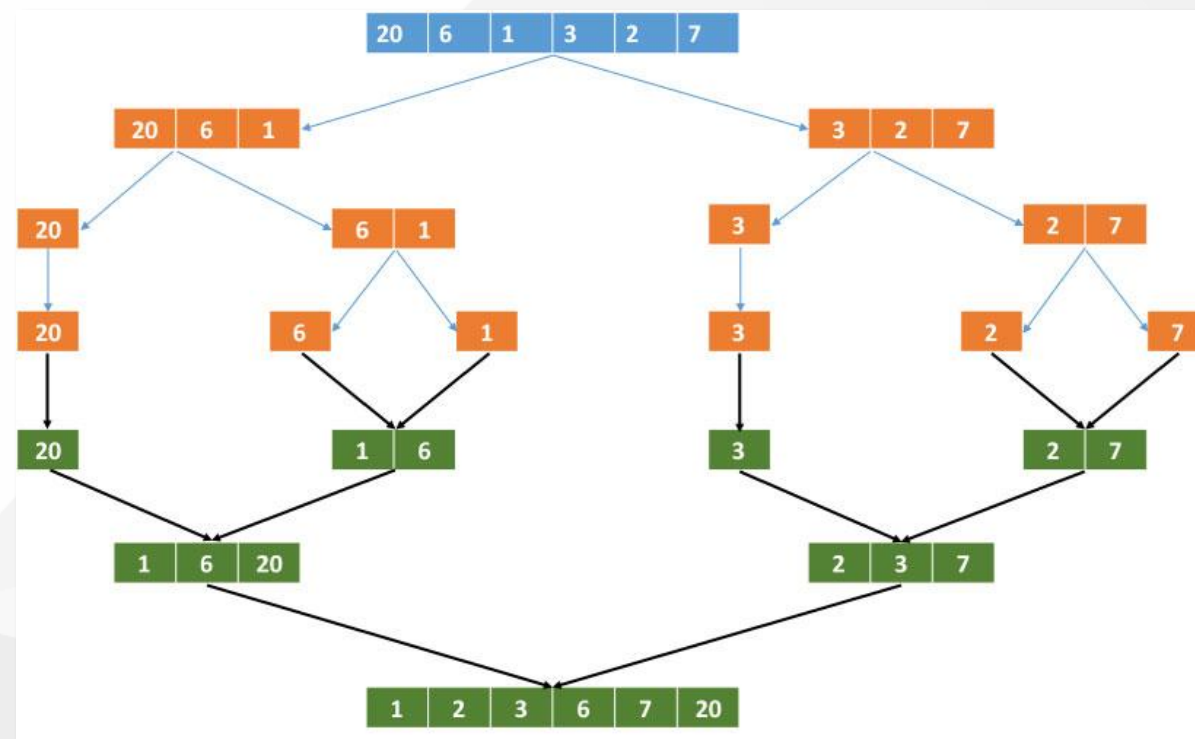
TRI PAR FUSION

En utilisant la technique Diviser pour régner, nous divisons un problème en sous-problèmes.

Lorsque la solution à chaque sous-problème est prête, nous «combinons» les résultats des sous-problèmes pour résoudre le problème principal.

La fonction triFusion divise à plusieurs reprises le tableau en deux moitiés (deux sous-tableaux) jusqu'à ce que nous atteignons un stade où nous essayons d'effectuer triFusion sur un sous-tableau de taille 1, c'est-à-dire debut == fin.

Après cela, la fonction de fusion entre en jeu et combine les tableaux triés dans un tableau plus grand jusqu'à ce que l'ensemble du tableau soit fusionné.



TRI PAR FUSION

L'algorithme de tri par fusion divise récursivement le tableau en deux jusqu'à ce que nous atteignons le cas de base d'un tableau avec 1 élément.

Après cela, la fonction de fusion récupère les sous-tableaux triés et les fusionne pour trier progressivement l'ensemble du tableau.

```
fonction triFusion(T, début, fin):  
    Si début > fin alors  
        retourne  
  
    # Trouvez le point milieu pour diviser le tableau en deux moitiés  
    milieu = (debut+fin)/2  
  
    # Appelez triFusion pour la première moitié du tableau:  
    triFusion(T, debut, milieu)  
  
    # Appelez triFusion pour la deuxième moitié du tableau:  
    triFusion(T, milieu+1, fin)  
  
    # Fusionnez les deux moitiés triées  
    fusion(T, debut, fin, milieu)
```

L'algorithme maintient trois pointeurs, un pour chacun des deux tableaux et un pour maintenir l'index actuel du tableau trié final.

- Comparer les éléments actuels des deux tableaux (T1[i] et T2[j])
- Copiez l'élément le plus petit dans le tableau trié
- Déplacer le pointeur de l'élément contenant un élément plus petit (i ou j)
- Copiez tous les éléments restants du tableau non vide



PERFORMANCE DES TRIS

LEGEND

TIME Complexity vs. SPACE Complexity

Good Fair Bad

Good Fair Bad

<BIG-O-CHEATSHEET>

www.bigocheatsheet.com

DATA STRUCTURE Operations

DATA Structure

TIME Complexity

SPACE Complexity

Average

Worst

Access **Search** **Insertion** **Deletion**

Worst

Array

Stack

Queue

Singly-Linked List

Doubly-Linked List

Skip List

Hash Table

Binary Search Tree

Cartesian Tree

B-Tree

Red-Black Tree

Splay Tree

AVL Tree

ARRAY SORTING Algorithms

ARRAY Algorithms

TIME Complexity

SPACE Complexity

Best

Average

Worst

Worst

Quicksort

Mergesort

Timsort

Heapsort

Bubble Sort

Insertion Sort

Selection Sort

Tree Sort

Shell Sort

Bucket Sort

Radix Sort

Counting Sort

Cubesort

Operations

Elements

$O(n!)$ $O(2^n)$ $O(n^2)$ $O(n \log n)$ $O(n)$ $O(1)$ $O(\log n)$

DATA Structure	Access	Search	Insertion	Deletion	TIME Complexity	SPACE Complexity
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	Good	Good
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Good	Good
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Good	Good
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Good	Good
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	Good	Good
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	Good	Good
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
Cartesian Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
Splay Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	Good	Good

ARRAY Algorithms	TIME Complexity	SPACE Complexity
Quicksort	$O(n \log(n))$	$O(\log(n))$
Mergesort	$O(n \log(n))$	$O(n)$
Timsort	$O(n)$	$O(1)$
Heapsort	$O(n \log(n))$	$O(1)$
Bubble Sort	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(1)$
Tree Sort	$O(n \log(n))$	$O(n)$
Shell Sort	$O(n \log(n)^2)$	$O(1)$
Bucket Sort	$O(n+k)$	$O(n)$
Radix Sort	$O(nk)$	$O(n+k)$
Counting Sort	$O(n+k)$	$O(k)$
Cubesort	$O(n)$	$O(n)$



CONCEPT

LA RÉCURSIVITÉ

C'EST QUOI ?

La récursivité propose une autre approche des traitements, plus séduisante, plus simple à écrire mais souvent plus complexe à concevoir.

La récursivité c'est quand une fonction s'appelle elle-même jusqu'à atteindre une condition d'arrêt. Elle s'arrête alors de s'appeler elle-même.

Cela signifie que la fonction utilise sa propre définition pour résoudre des instances plus petites (ou plus simples) du même problème, jusqu'à atteindre un cas de base où la solution peut être directement déterminée sans appel récursif.

DÉFINITION

Voici les éléments clés de la récursivité en algorithmique :

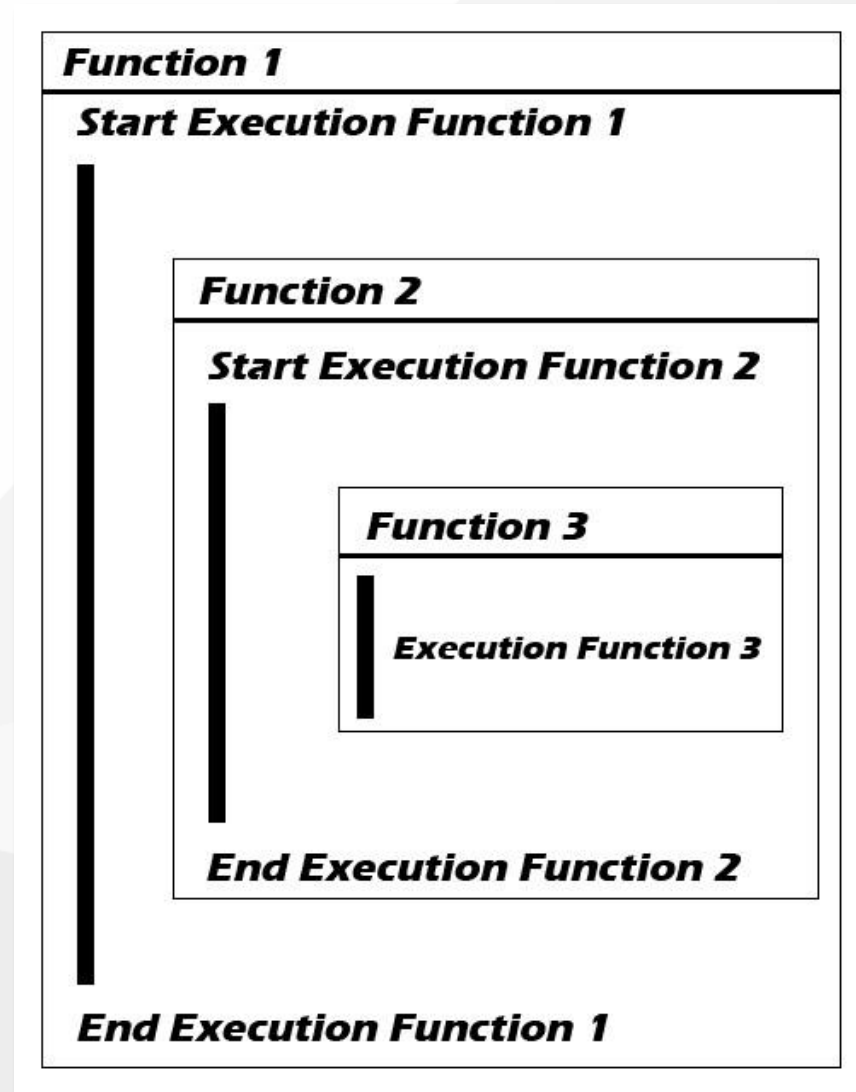
- **Cas de base** : C'est la condition qui spécifie le point d'arrêt de la récursion. Lorsque cette condition est vérifiée, la fonction récursive cesse de s'appeler elle-même et retourne une valeur ou effectue une action spécifique.
- **Cas récursif** : C'est la partie de la fonction récursive où elle s'appelle elle-même avec des arguments modifiés.
 - Cette étape divise le problème initial en sous-problèmes plus petits, qui sont résolus de manière récursive.
- **Progression vers le cas de base** : À chaque appel récursif, les arguments doivent être modifiés d'une manière qui se rapproche du cas de base.
 - Sinon, la récursion ne s'arrêtera jamais et entraînera une boucle infinie.
- **Résolution des sous-problèmes** : Les appels récursifs résolvent les sous-problèmes plus petits jusqu'à atteindre le cas de base. Les résultats des sous-problèmes sont ensuite utilisés pour résoudre le problème global.

FONCTIONNEMENT

Quand une fonction s'appelle elle-même, la fonction enfant fait partie de l'exécution de la fonction parent.

Autrement dit, l'exécution de chaque fonction est imbriquée de plus en plus profondément à chaque appel.

Sans condition d'arrêt, ça devient très vite un problème.



EXEMPLE SIMPLE

Compter à l'envers en partant de 2.

Avec une simple boucle, cela sera tout à fait possible :

```
// Programme Compter_a_lenvers
// Auteur : Jérôme BOEBION
// Description : Compter à l'envers en partant de 2
Programme Compter_a_lenvers()

Début
    Pour i Allant de 2 à 0 Faire
        Ecrire(i)
        i <- i - 1
Fin
```

Mais on pourrait aussi utiliser la récursivité à l'aide d'une fonction...

Pour utiliser la récursivité, il y a donc 3 grandes étapes pour chaque algorithme récursif

1. La condition d'arrêt
2. La résolution du problème
3. L'appel récursif

Dans notre exemple :

1. La condition d'arrêt : c'est quand le décompte est à zéro.
Obligatoire sinon on ne sortira pas de la récursivité
2. La résolution du problème, Ici c'est afficher le nombre en cours
3. L'appel récursif. c'est de réduire le problème par un problème plus petit. Ici, l'appel suivant est de réduire le problème pour approcher la condition d'arrêt soit arriver à zéro. Donc rappeler la fonction en réduisant le nombre de - 1

Voyons ce que cela donnera...

RÉSULTAT

1. La condition d'arrêt : [ligne 12](#)
2. La résolution du problème : [ligne 13 et 15](#)
3. L'appel récursif : [ligne 16](#)

```
1 // Programme Compter_a_lenvers
2 // Auteur : Jérôme BOEBION
3 // Description : Compter à l'envers en partant de x
4 Programme Compter_a_lenvers(nombre : Entier)
5
6 Variables
7     saisie : Entier
8
9 Fonctions
10 Fonction Compte(nombre : Entier)
11     Début
12         Si nombre = 0 Alors
13             Ecrire(nombre)
14         Sinon
15             Ecrire(nombre)
16             Compte(nombre - 1)
17         FinSi
18     Fin
19
20 Début
21     Ecrire("Entrez un nombre ?")
22     Lire(saisie)
23
24     Compte(saisie)
25 Fin
```

AUTRE EXEMPLE

Écrivez une fonction récursive qui calcule la somme des chiffres d'un nombre entier donné.

Par exemple, si le nombre est 123, la somme des chiffres serait $1 + 2 + 3 = 6$.

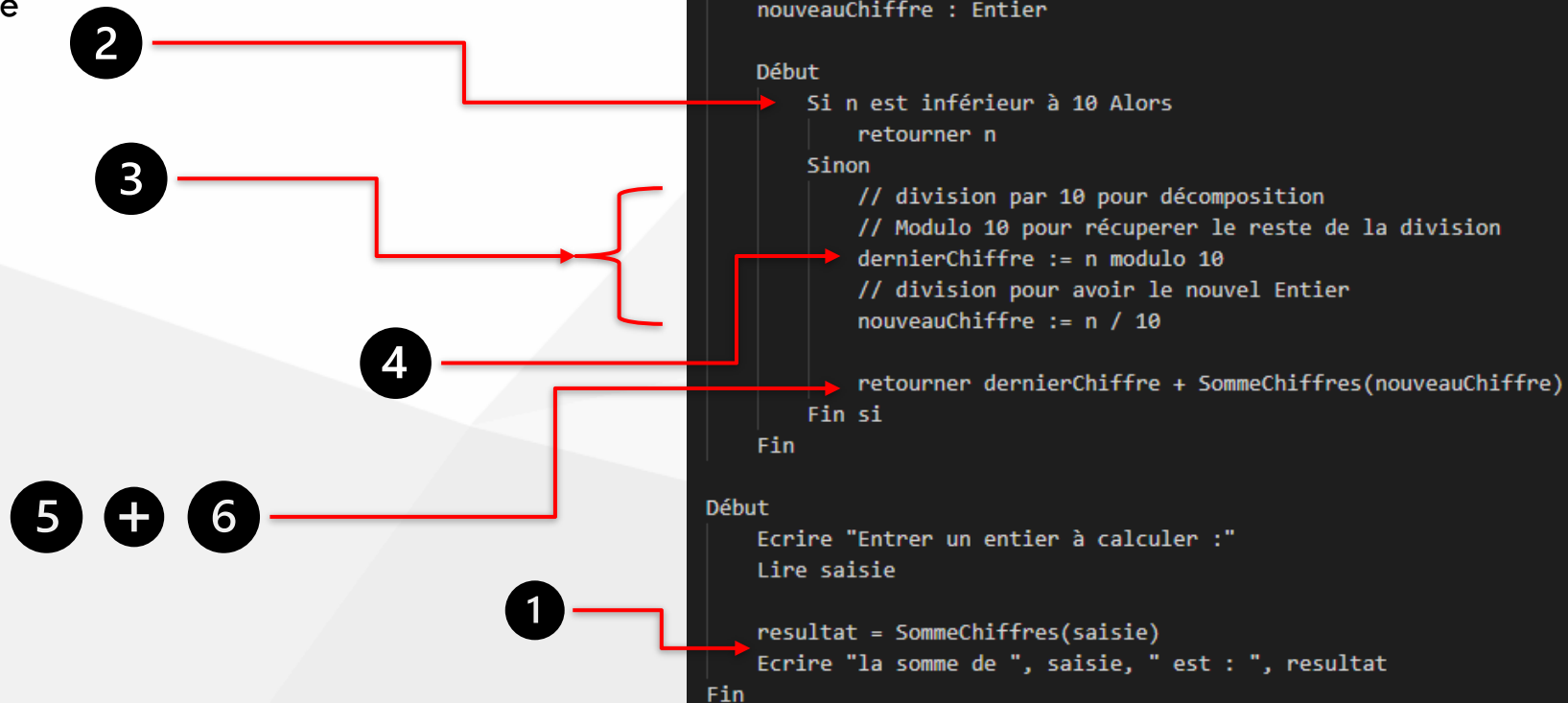
Instructions :

1. Définissez une fonction récursive appelée "sommeChiffres" qui prendra un entier en entrée.
2. Vérifiez si l'entier est inférieur à 10. Si c'est le cas, cela signifie que c'est un chiffre unique, donc retournez simplement cet entier.
3. Sinon, divisez l'entier par 10 pour obtenir un nouveau nombre avec un chiffre de moins à chaque appel récursif.
4. Ajoutez le chiffre le plus à droite de l'entier à la somme des chiffres.
5. Appelez récursivement la fonction "sommeChiffres" avec le nouveau nombre obtenu.
6. Ajoutez la valeur retournée par l'appel récursif à la somme des chiffres.
7. Retournez la somme des chiffres.

- **Cas de base** : (2) notre cas de base sera donc si j'ai un chiffre inférieur à 10, je dois retourner le chiffre lui-même et arrêter la récursivité.
- **Cas récursif** : (3) Dans notre fonction récursive, on aura donc pour objectif de diviser l'entier par 10 pour obtenir un nouvel entier.
- **Progression vers le cas de base** : (4) À chaque appel récursif, par la division, on récupère un nombre qui servira dans le calcul de la somme demandée. C'est ici qu'on détermine les nouvelles données.
- **Résolution des sous-problèmes** : (5 et 6), à chaque appel récursif, on fera la somme de chaque nombre obtenu par appel récursif jusqu'à aller au cas de base.

SOLUTION

Ecriture de l'algorithme



DÉTAILS

Reprenons l'exemple 123

A partir de notre pseudo-code.

1. L'utilisateur saisie 123
2. Résultat = SommeChiffres(saisie)
3. Résultat = 3 + 2 + 1
4. La somme de 123 est : 6

1

Lancement de ma récursivité :

- résultat = SommeChiffres(123)
 - 1^{er} appel
 - dernierChiffre = 3
 - nouveauChiffre = 12
 - retourner 3 + SommeChiffres(12)
 - 2^{ème} appel
 - dernierChiffre = 2
 - nouveauChiffre = 1
 - retourner 3 + 2 + SommeChiffres(1)
 - 3^{ème} appel
 - 1 < 10 donc retourne 1
 - retourner 3 + 2 + 1

EXEMPLE

la suite de Fibonacci pour un nombre entier N se définit comme la relation de récurrence :

- $F(N) = F(N-1) + F(N-2)$ pour $N \geq 2$ avec $F(0) = 0$ et $F(1) = 1$

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	...	F_n
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	...	$F_{n-1} + F_{n-2}$



PSEUDO-CODE FIBONACCI

Voici ce que le programme en pseudo-code pourrait donner.

```
Programme SommeFibonacci

// déclaration
Variable saisie : Entier
Variable resultat : Entier

Fonction Fibonacci(n : Entier) : Entier
    Début
        Si n est égal à 0 Alors
            retourner 0
        Sinon
            si n est égal à 1 Alors
                retourner 1
            Sinon
                retourner Fibonacci(n-1) + Fibonacci(n-2)
            Fin Si
        Fin Si
    Fin

Début
    Ecrire "Entrer un entier à calculer :"
    Lire saisie

    resultat = Fibonacci(saisie)
    Ecrire "Fibonacci de ", saisie, " est : ", resultat
Fin
```

DÉCOMPOSITION

1. **Cas de base** : Si n est égal à 0 on retourne 0 et si n est égal à 1, on retourne 1
2. **Cas récursif** : c'est la formule de Fibonacci qui dit que $F(n) = F(n-1) + F(n-2)$

Exemple si $n = 2$ alors

$$F(2) = F(1) + F(0)$$

$$F(2) = 1$$

si $n = 3$ alors

$$F(3) = F(2) + F(1)$$

$$F(3) = F(1) + F(0) + F(1)$$

$$F(3) = 2$$

```
Fonction fibonacci(n)
  Si n est égal à 0 Alors
    Retourner 0
  Sinon Si n est égal à 1 Alors
    Retourner 1
  Sinon
    Retourner fibonacci(n-1) + fibonacci(n-2)
  Fin Si
Fin Fonction
```

3. **Progression vers le cas de base** : l'objectif est de se rapprocher par appel récursif vers $F(0)$ et $F(1)$. C'est ici que nous allons modifier les données pour relancer l'appel récursif pour aller au cas de base.
4. **Résolution des sous-problèmes** : appel récursif avec les nouvelles données.



CONCEPT

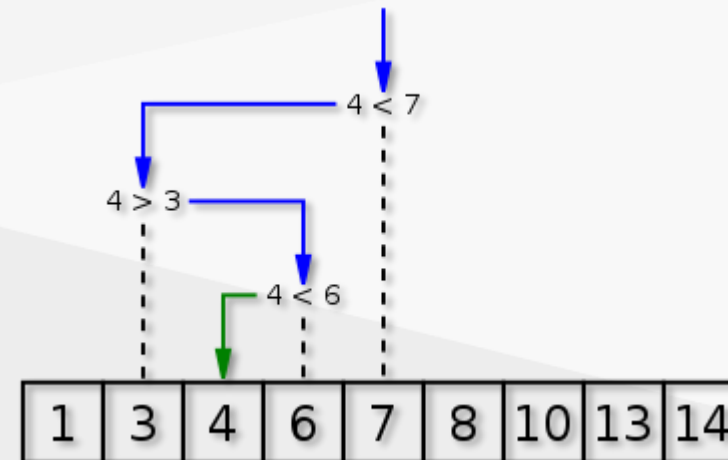
LA RECHERCHE DICHOTOMIQUE

RECHERCHE DICHOTOMIQUE

Cet algorithme permet de ranger un élément à sa place ou de le trouver dans une **liste triée** de manière très rapide.

Le principe est le suivant : comparer l'élément avec la valeur de la case au milieu du tableau. Si les valeurs sont égales, la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente.

Visualisation d'une recherche dichotomique, où 4 est la valeur recherchée.



RECHERCHE DICHOTOMIQUE

Voici l'algorithme simple

```
fonction rechercheDichotomique(tableau, valeurCherchee)
    debut ← 0
    fin ← longueur(tableau) - 1

    tant que debut ≤ fin faire
        milieu ← (debut + fin) / 2 // division entière

        si tableau[milieu] == valeurCherchee alors
            retourner milieu // index trouvé
        sinon si tableau[milieu] < valeurCherchee alors
            debut ← milieu + 1
        sinon
            fin ← milieu - 1
        fin si
    fin tant que

    retourner -1 // valeur non trouvée
fin fonction
```

RECHERCHE DICHOTOMIQUE

Écriture récursive

```
RechDichoRecur(élément, liste):
    taille := longueur de liste
    milieu := taille / 2
    si liste[milieu] = élément alors
        renvoyer milieu
    fin si
    si liste[milieu] > élément alors
        renvoyer RechDichoRecur(élément, liste[1,milieu-1])
    sinon
        renvoyer milieu + RechDichoRecur(élément, liste[milieu+1,taille])
    Fin si
```

Note : la liste est préalablement triée

Démonstration

rechercher 7

7<30, rechercher dans la partie gauche du milieu

2	7	8	20	25	30	33	39	45	50
---	---	---	----	----	----	----	----	----	----

7<8, rechercher dans la partie gauche du milieu

2	7	8	20	25	30	33	39	45	50
---	---	---	----	----	----	----	----	----	----

7= élément au milieu
Retourner indice du milieu

2	7	8	20	25	30	33	39	45	50
---	---	---	----	----	----	----	----	----	----

Retourner 1

RECHERCHE DICHOTOMIQUE

Voici la version testée dans AlgoBox

Résultats

```
***Algorithme lancé***
nombre à verifier ?
Entrer valeurRecherchee : 5
La valeur existe
***Algorithme terminé***
```

Résultats

```
***Algorithme lancé***
nombre à verifier ?
Entrer valeurRecherchee : 18
la valeur n'existe pas
***Algorithme terminé***
```

```
FONCTIONS_UTILISEES
VARIABLES
    max EST_DU_TYPE NOMBRE
    min EST_DU_TYPE NOMBRE
    indice EST_DU_TYPE NOMBRE
    valeurRecherchee EST_DU_TYPE NOMBRE
    dico EST_DU_TYPE LISTE
    flag EST_DU_TYPE NOMBRE
DEBUT_ALGORITHME
    //initialisation des variables
    dico[0] PREND_LA_VALEUR 1:2:3:4:5:6:7:8:9:10
    flag PREND_LA_VALEUR 0
    AFFICHER "nombre à verifier ?"
    LIRE valeurRecherchee
    max PREND_LA_VALEUR dico.length-1
    min PREND_LA_VALEUR 0
    TANT_QUE (min <= max ET flag == 0) FAIRE
        DEBUT_TANT_QUE
            //determine le milieu
            indice PREND_LA_VALEUR floor((max+min)/2)
            //Si la valeur est sur l'indice du milieu c'est trouvé
            SI (dico[indice] == valeurRecherchee) ALORS
                DEBUT_SI
                    AFFICHER "La valeur existe"
                    flag PREND_LA_VALEUR 1
                FIN_SI
            SINON
                DEBUT_SINON
                    //Sinon on modifie min ou max selon si la valeur se trouve dans le sous ensemble
                    gauche ou droite
                    SI (dico[indice] < valeurRecherchee) ALORS
                        DEBUT_SI
                            min PREND_LA_VALEUR indice+1
                        FIN_SI
                    SINON
                        DEBUT_SINON
                            max PREND_LA_VALEUR indice-1
                        FIN_SINON
                    FIN_SINON
                FIN_TANT_QUE
            //dans le cas où flag n'a pas bougé c'est que valeur n'existe pas
            SI (flag == 0) ALORS
                DEBUT_SI
                    AFFICHER "la valeur n'existe pas"
                FIN_SI
            FIN_ALGORITHME
```

QUELQUES EXEMPLES

Algorithme par la pratique.



PARTIE 4

Exercice 4.1

Ecrire un algorithme qui déclare un tableau de 9 notes, dont on fait ensuite saisir les valeurs par l'utilisateur.

Enfin après la saisie on affichera la moyenne des notes.

Exercice 4.2

Ecrire un algorithme qui trie un tableau dans l'ordre croissant. Et tester l'algorithme en mode pas à pas.

Note : écrire deux versions :

1 - en utilisant le tri par insertion

2 - en utilisant le tri à bulles

PARTIE 4

Exercice 4.3

Ecrire l'algorithme qui recherche un nombre saisi au clavier dans un dictionnaire. Et tester en mode pas à pas.

Note : Le dictionnaire est supposé être codé dans un tableau préalablement rempli et trié.

Quel Algorithme allez-vous utiliser ?

Exercice 4.4

Soit un tableau T à deux dimensions (12,8) préalablement rempli de valeurs numériques.

Ecrire un algorithme qui recherche la plus grande valeur au sein de ce tableau

PARTIE 5

Exercice 5.1

Nombre d'occurrences d'un élément x dans un tableau

Soit un tableau contenant une liste de N nombres entiers dont la valeur est comprise entre 0 et 100.

Ecrivez l'algorithme en utilisant qu'un seul tableau permettant de trouver le nombre de fois que x se trouve dans le tableau

Exercice 5.2

Modifiez l'exercice précédent pour utiliser deux tableaux. Le 1^{er} contiendra les données et le 2^{eme}, les occurrences pour chaque nombre entier. On pourra utiliser l'entier à traiter comme indice du deuxième tableau.

Ex :

tableau_données [1, 2, 3, 2, 4, 1]

tableau_occurences [0, 2, 2, 1, 1]

PARTIE 6 (DIFFICILE) (RÉCURSIVITÉ)

Exercice 6


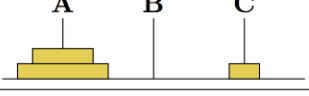
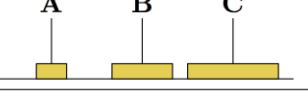


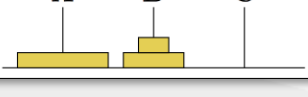
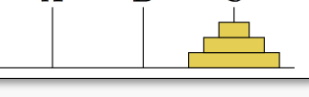

Les tours de Hanoï (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

On ne peut déplacer plus d'un disque à la fois.

On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Ecrivez l'algorithme de ce jeu dont voici un exemple :

Mouvement	Position	Mouvement	Position
Position initiale			
1 : A vers C		4 : A vers C	
2 : A vers B		5 : B vers A	
3 : C vers B		6 : B vers C	
		7 : A vers C	

MERCI !

Jérôme BOEBION
Concepteur Développeur d'Applications
Version 1.2 - révision 2024

