Institut National Polytechnique
Félix HOUPHOUET-BOIGNY

LARIT

Cours d'informatique

Algo en Prépa

Classes Préparatoires Scientifiques 1

Initiation à L'algorithme

Prépa
MPSI & BCPST 1
2018 - 2019

Daniel KPO LOUA
Enseignant-Chercheur en informatique à l'INP-HB
Département : Maths – Info (DMI)
kpoloua@gmail.com / 00225 0707 774384

CM : 10 H | TD : 05 H

Septembre 2021

A Celui qui peut tout
Et qui me rend capable de tout,
Mon bien-aimé Père !

Table des matières

CHAPITRE 3

Notion d'algorithme

« *Un langage de programmation est une convention pour donner des ordres à un ordinateur. Ce n'est pas censé être obscur, bizarre et plein de pièges subtils. Ça, ce sont les caractéristiques de la magie.* »

Dave Small

1. Concernant ce cours

Le but de ce cours est d'étudier comment on peut résoudre un problème en utilisant **le Langage de Description des Algorithmes** suivant l'approche **Bertini** et **Tallineau**. Il va donc falloir étudier les principes fondamentaux de la programmation, base valable quel que soit le langage utilisé.

A la fin de ce cours vous serez capable de :

- **analyser,** spécifier et modéliser de manière rigoureuse une situation ou un problème, indépendamment d'un langage de programmation ;
- **expliquer** le fonctionnement d'un algorithme ;
- **déterminer** la trace d'un algorithme ;
- **justifier** qu'une itération (ou boucle) produit l'effet attendu au moyen d'un invariant ;
- **démontrer** qu'une boucle se termine effectivement, modifier un algorithme existant pour obtenir un résultat différent ;
- **décomposer** un problème complexe en sous-problèmes relativement simples

1.1. Compétences requises de l'étudiant

Faudrait-il que l'étudiant soit :

- Un informaticien,
- Un matheux,
- Un Prépa….

pour comprendre ce cours d'algorithme ? Non, je ne pense pas. La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs :

1. Il faut avoir une certaine intuition, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, si l'on y tient, qu'intervient la forme « d'intelligence » requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, et j'insiste sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».

2. Il faut être **méthodique** et **rigoureux**. En effet, chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut **systématiquement** se mettre mentalement à la place de la machine qui va les exécuter, armé d'un papier et d'un crayon, afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas la moindre once d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

1.2. Un petit conseil

Pour mieux bénéficier de ce cours, il vous est fortement recommandé comme d'ailleurs, de vous exercer dans un rythme maintenu. Ecrire deux algorithmes chaque soir avant de vous endormir. Une période d'un mois serait largement suffisante pour déceler le mythe des algorithmes. Ainsi parviendrez-vous à faire face à tout algorithme quel que soit son degré de complexité.

1.3. L'algorithmique

1.3.1. Histoire

C'est un mathématicien Perse du 8ème siècle, Al-Khawarizmi, qui a donné son nom à la notion d'algorithme. Son besoin était de traduire un livre de mathématiques venu d'Inde pour que les résultats et les méthodes exposés dans ce livre se répandent dans le monde Arabe puis en Europe. Les résultats devaient donc être compréhensibles par tout autre mathématicien et les méthodes applicables, sans ambiguïté.

Si l'origine du mot algorithme est très ancienne, la notion même d'algorithme l'est plus encore : on la sait présente chez les Babyloniens, 1800 ans avant JC.

1.3.2. L'algorithme

- **Lire attentivement ce texte :**

Le mot attiéké est une déformation du mot *'adjèkè'* de la langue ébrié parlée dans le sud de la Côte d'Ivoire. À l'origine (et parfois encore aujourd'hui), les femmes ébriés ne confectionnent pas de la même manière l'attiéké qu'elles vendent avec celui qui est consommé par leur propre ménage. Par conséquent, elles qualifiaient d'adjèkè le produit vendu pour le commerce ou pour la vente, afin de marquer la différence avec le produit consommé à la maison (Ahi). Ce sont ensuite les transporteurs bambaras qui ont propagé ce mot le faisant passer à 'atchèkè'. Les colons français (certainement pour motif

d'esthétisme à l'écriture) écrivirent 'attiéké'; mais, dans la rue, on prononce souvent 'tch(i)éké', avec amuïssement du a initial.
Spécialité culinaire de certains peuples lagunaires (Ebrié, Adjoukrou, Alladian, Abidji, Avikam, Attié, Ahizi) du Sud de la Côte d'Ivoire, l'attiéké est traditionnellement produit par les femmes, des équipes constituées dans le village se groupant pour la production. Sa consommation est tellement forte que des usines ont été construites pour le fabriquer. (Source : wikipédia.org)

L'une des variétés de l'attiéké est le Garba. Pour sa préparation vous aurez besoin des ingrédients suivants :

- ✓ 1 boule d'attiéké, ou 2 sachets d'attiéké déshydraté
- ✓ 1 cube d'assaisonnement
- ✓ 3 cuillères à soupe d'huile chaude
- ✓ 1 petit oignon coupé en dés
- ✓ 1 tomate coupée en petits dés
- ✓ 1 piment vert finement haché
- ✓ Du sel et du poivre

Pour l'accompagnement soit du poulet ou du poisson thon (humm !)

- ✓ 4 tranches conséquentes de thon frais
- ✓ 1 bonne poignée de farine
- ✓ Du sel et du poivre

Passons maintenant à la préparation

L'attiéké :

☞ Si vous utilisez l'attiéké déshydraté :
Suivez toutes les instructions inscrites sur le paquet.
Une fois l'attiéké chaud, saupoudré-le d'assaisonnement Maggi
Arrosez l'attiéké de l'huile chaude. Salez, poivrez, puis mélangez vigoureusement le tout.
Disposer l'attiéké selon vos convenances et disposer l'oignon, la tomate en dés et le piment par-dessus le plat.

☞ Si vous utilisez l'attiéké en boule :
Dans un saladier qui va dans le micro-ondes, émiettez grossièrement l'attiéké que vous humidifiez avec 2 à 3 cl d'eau 2 minutes dans le micro-onde suffiront largement. Autrement émiettez grossièrement l'attiéké dans un panier à vapeur.
L'attiéké se prépare alors comme le couscous de blé. Une fois l'attiéké chaud, saupoudrez-le d'assaisonnement Maggi.
Arrosez l'attiéké de l'huile chaude. Salez, poivrez, puis mélangez vigoureusement le tout.

Disposer l'attiéké selon vos convenances et disposer l'oignon, la tomate en dés et le piment par-dessus le plat.

Le poisson :

Salez, poivrez et farinez vigoureusement les morceaux de thon.
Faites-les frire pendant une dizaine de minutes dans une huile bien chaude.
Enfin déposez simplement le poisson par-dessus l'attiéké.
Le Garba est prêt ! ça se mange avec la main !

Avez-vous déjà ouvert un livre de recettes de cuisine ou fait la cuisine ? (si non retournez au texte ci-dessus) Avez-vous déjà déchiffré un mode d'emploi traduit directement de l'Anglais pour faire fonctionner un appareil ? Si oui, sans le savoir, vous avez déjà exécuté des algorithmes.

Plus fort encore : avez-vous déjà indiqué un chemin à un touriste égaré ou à un inconnu égaré dans votre quartier ? Avez-vous fait chercher un objet à quelqu'un par téléphone ? Ecrit une lettre anonyme stipulant comment procéder à une remise de rançon ? Si oui, vous avez déjà fabriqué – et fait exécuter – des algorithmes.

- Un algorithme est un ensemble de règles opératoires propres à un calcul ou l'enchaînement des actions nécessaires à l'accomplissement d'une tâche.

- Un algorithme est un procédé reprenant un ensemble de suites élémentaires d'actions à exécuter afin de résoudre un problème à partir des données de départ et d'arriver à un résultat final déterminé.

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement dans un ordre bien précis, conduit à un résultat donné, solution à un problème posé.

En résumé, il doit être bien clair que cette notion d'algorithme dépasse, de loin, l'informatique et les ordinateurs. La création d'un algorithme nécessite un vocabulaire partagé, des opérations de base maîtrisées par tous et de la précision.

1.3.3. L'algorithmique

C'est la logique d'écrire des algorithmes. Pour pouvoir écrire des algorithmes, il faut connaître la résolution manuelle du problème, connaître les capacités de l'ordinateur en termes d'actions élémentaires qu'il peut assurer et la logique d'exécution des instructions.

L'algorithme est indépendant des langages de programmation : si vous prenez une méthode de tri d'un tableau, par exemple le tri bulle, vous pouvez l'écrire dans la plupart des langages de programmation usuels. L'algorithmique va donc s'attacher à l'étude de différentes méthodes permettant de résoudre un problème donné sans tenir compte des particularités de tel ou tel langage de programmation. À ce titre l'algorithmique se veut universelle.

Le schéma usuel de résolution d'un problème sera donc :

1.3.4. La spécification de l'algorithme

Spécifier un problème revient à expliciter les *informations pertinentes* pour une modélisation algorithmique, notamment les données (les entrées) et la sortie (résultat), puis à formuler les relations qui les caractérisent (le traitement). Il existe plusieurs façons de spécification d'un algorithme

- Le langage naturel ;
- Le Langage de Description Algorithmique (LDA) que nous utiliserons dans ce cours ;
- La forme de graphe (arbre binaire, ou autres)

Il n'existe pas hélas de langages de description des algorithmes universellement reconnus. Nous proposerons ici une syntaxe particulière, qui ressemble aux langages algorithmiques usuellement utilisés.

NB: Une information est une connaissance supplémentaire sur une donnée.
Ex : Mfoutu a **20 ans** fait référence à l'âge.

Une donnée est la représentation conventionnelle d'une information permettant d'en faire le traitement automatique1. Ex : 20.

Une donnée est généralement désignée par un nom et son type. Ex : âge = 20 de type entier naturel positif.

1.3.5. Les caractéristiques et les propriétés d'un algorithme

♦ **Les caractéristiques d'un algorithme**

La validité

La validité d'un algorithme est son aptitude à réaliser exactement la tâche pour laquelle il a été conçu.

La robustesse

La robustesse d'un algorithme est son aptitude à se protéger de conditions anormales d'utilisation.

La réutilisabilité

La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu.

La complexité

La complexité d'un algorithme est le nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu.

L'efficacité

L'efficacité d'un algorithme est son aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute.

Les propriétés d'un algorithme

Un algorithme doit avoir les propriétés suivantes :

- Avoir un nombre fini d'étapes
- Doit fournir un résultat
- Avoir un comportement déterministe

Chaque opération doit être

- **Définie** rigoureusement et sans ambigüité
- **Effective**, c'est-à-dire réalisable par la machine

1.4. Le Langage de Description des Algorithmes

Le LDA est un pseudo-langage qui permet d'élaborer un algorithme. Il est beaucoup plus proche du langage humain et est facilement compréhensible par l'homme. L'avantage d'un tel langage est de pouvoir être facilement transcrit dans un langage de programmation.

Le LDA donne une vue générale des langages de programmation informatique (logique et fonctionnement).

Les caractéristiques du LDA

- Le LDA utilise un ensemble de mot clés et de structures permettant de décrire de manière complète et claire, l'ensemble des opérations à exécuter sur des données pour obtenir un résultat.
- L'avantage d'un tel langage est de pouvoir facilement être transcrit dans un langage de programmation structuré tels que pascal, C ou Python…
- Le LDA permet également d'écrire un algorithme sans avoir au préalable connaissance d'un langage de programmation informatique. Il permet de représenter notre solution sous un format indépendant d'un langage de programmation informatique et de faciliter sa compréhension par un grand public.
- Il n'existe pas de normes ou syntaxes universelles pour les éléments du LDA. Ce n'est pas un langage standardisé. La réalisation d'un algorithme en LDA peut différer d'un individu à un autre, mais la logique et les principes sont les mêmes.
- Tous les éléments existants dans un langage de programmation informatique peuvent ne pas exister en LDA. Il faut souvent nous-même définir des expressions pour les représenter.

- Le LDA n'est certes pas indispensable pour la réalisation d'un programme informatique, mais demeure très importante pour une meilleure structuration et compréhension de notre solution ou algorithme avant sa programmation.

1.5. L'analyse du problème

Avant de résoudre un problème, il faut l'analyser avec précision. Cette réflexion passe par certaines étapes incontournables

1.5.1. Définition du problème

On ne peut pas résoudre un problème si on ne l'a pas défini précisément ! Il faut donc fixer le plus clairement possible les différentes fonctionnalités du programme.

Il s'agit de lire plusieurs fois le problème afin de bien le cerner et d'y détecter un certain nombre de routines. C'est l'étape la plus importante dans l'algorithmique car une fois le problème mal compris, la solution que l'on proposerait ne serait plus contextuel et valide.

1.5.2. L'indentification des informations

Il faut définir ici quelles sont les données au point de départ, quelles données doivent être sauvegardées et quelles sont les données qui vont apparaitre de façon intermédiaire.

- **Les données ou les entrées**

Ce sont les informations que fait ressortir le problème. Elles sont présentes dans le sujet ou le libellé du problème et pour les débusquer il suffit de lire le problème et de le comprendre.

- **Les intermédiaires**

Ces données interviennent pour combler un besoin, soit pour stocker temporairement une valeur intermédiaire ou parcourir une liste de variables ou stocker un résultat intermédiaire.

- **Les sorties ou résultats**

Ce sont les informations que doit retourner l'algorithme à un utilisateur. C'est la solution au problème posé, le résultat tant attendu !

1.5.3. La spécification des résultats

Il faut définir les différentes étapes de notre programme depuis la saisie des données jusqu'à l'affichage du résultat. On peut définir un prototype qui décrira point par point comment doit fonctionner notre programme.

1.5.4. Analyse des données

Avant de résoudre le problème,

- ☞ il faut définir la nature des différentes données que manipulera notre programme ;
- ☞ Il faut définir ici quelles sont les données au point de départ et quelles données doivent être sauvegardées.

Il serait mieux de dresser un tableau pour faire un bilan.

1.5.5. Le corps de l'algorithme (Le traitement).

C'est la phase technique où il faut proposer une méthode pratique permettant de résoudre notre problème. On parle du traitement.

1.5.6. La validation de l'algorithme

Après l'écriture de l'algorithme, il est fortement recommandé de le tester en vue de le valider. Ce test se fait pratiquement à la main à l'aide d'un brouillon de calcul.
Les principales étapes sont :

- ☞ Dessiner les cases mémoires de toutes les variables ;
- ☞ Exécuter l'algorithme en modifiant le contenu de chaque variable pour chaque instruction sur celle-ci ;
- ☞ Vérifier si le résultat final correspond parfaitement à la solution du problème posé.

1.6. Les déclarations

1.6.1. Les arbres programmatiques (AP)

Cette méthode permet de représenter en arbre structuré les différents algorithmes du programme. Le programme (racine de l'arbre) peut se décomposer en sous-programmes (sous niveaux) toujours organisés en deux parties :

- Les Déclarations (variables et constantes : **Données**)
- Les Instructions (traitements sur les données : **Opération**)

```
Déclaration
DEBUT
   - instruction(s)
FIN
```

Un algorithme va utiliser des variables. Une variable peut être vue comme une zone mémoire réservée permettant de stocker une valeur. Toute variable utilisée dans la partie « instruction » de l'algorithme doit faire l'objet d'une déclaration préalable. La phase de déclaration doit s'accompagner d'une réflexion sur les données manipulées par notre algorithme. Quelle sera leur nature précise ? Leur valeur sera-t-elle variable ? Loin d'être une contrainte, la phase de déclaration est une aide pour guider le programmeur dans sa réflexion.

1.6.2. Les constantes

Définition

Une constante permet d'attribuer un nom à une valeur ou une expression fixe relativement complexe à retranscrire ou relativement longue. Elle permet également une meilleure lisibilité du code.

La déclaration d'une constante permet une modification simplifiée d'une valeur reprise à maintes reprises au sein du code. L'utilisation des constantes permettra d'améliorer la lisibilité de notre algorithme et sa généralisation.

Déclaration

Pour déclarer une constante on utilise le mot réservé (au LDA) CONST ou Const.
La syntaxe générale de cette déclaration se présente sous la forme :

CONST Nom_de_la_constante ← valeur : **Type**

Exemple :

```
CONST pi ← 3.14 : REEL
CONST ttl ← "Message à afficher régulièrement" : CHAINE
CONST Oui ← VRAI : BOOLEEN
CONST Resu ← pi*2+5*(15+4) : REEL
```

Pour des raisons de facilité il est préférable d'utiliser des noms de constantes plus ou moins courts et représentatifs de leur contenu.
Il est possible d'utiliser des déclarations multiples.

```
CONST pi ← 3.14 : REEL
      Ttl ← "Message à afficher" : CHAINE
      Oui ← Vrai : BOOLEEN
      Resu ← pi*2+5*(5+4) : REEL
```

1.6.3. Les variables

Définition

Une variable est une zone réservée de la mémoire (RAM) permettant de stocker une valeur. Pour atteindre cette zone mémoire on lui attribue **un nom**. La valeur ainsi stockée par la variable pourra être modifiée au cours de l'exécution du code. Pour pouvoir réserver la place nécessaire à la valeur, il va falloir déclarer **le type** du contenu et éventuellement sa taille.

Déclaration

Pour déclarer une variable on utilise le mot réservé (au LDA) de VAR ou Var.
La syntaxe générale de cette déclaration se présente sous la forme.

```
VAR Nom_de_la_variable : Type
```

```
VAR
      Nom_de_la_variable1 : Type1
      Nom_de_la_variable2 : Type2
      Nom_de_la_variable3, Nom_de_la_variable4 : Type3
```

Nom_de_la_variable représente le nom que l'on attribue à la variable.

Type permet de spécifier l'ensemble des valeurs dans lequel la valeur de la variable pourra varier. Il permet également de déterminer la dimension de la zone mémoire à réserver et le type d'opérations que pourra supporter la valeur.

On peut classer les différents types en catégories. Ainsi on parle de type primitif, il s'agit d'un type qui existe de fait avec le langage de programmation, il est défini en même temps que le langage.

On parle également de type scalaire, il s'agit d'un type qui possède un précédent (hormis la première valeur qui est la plus petite et un suivant hormis la dernière valeur qui est la plus grande) et est représentée ou codée par un entier dans la machine.

1.6.4. Les types

On dispose de quatre types primitifs, tous ordonnés mais dont 3 sont scalaires et 1 non.

- **Scalaire :** on trouve suivant ou précédant (ex. : ...5, **6**, 7...)
- **Primitif** : on trouve le plus petit ou le plus grand (ex. : 5,78 < 19,72)
- **ENTIER** (scalaire - primitif) Exemple : Var Max : Entier
- **REEL** (non scalaire - primitif) Exemple : Var Point : Reel
- **CARACTERE** (scalaire - primitif) Exemple : Var Initiale : Caractere

!"#$%&'()*+,./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc defghijklnopqrstuvwxyz{|}~

- **BOOLEEN** (scalaire - primitif) Exemple : Var Reussite : Booleen (deux états : Vrai ou Faux | 0 OU 1)
- **ALPHANUMERIQUE** (chaine de caractère) // Var Message : Alphanumérique

Voici un tableau représentatif de quelques types les plus utiliser en algorithme et programmation :

Type numérique	Plage
Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3{,}40\text{x}10^{38}$ à $-1{,}40\text{x}10^{45}$ pour les valeurs négatives $1{,}40\text{x}10^{-45}$ à $3{,}40\text{x}10^{38}$ pour les valeurs positives
Réel double	$1{,}79\text{x}10^{308}$ à $-4{,}94\text{x}10^{-324}$ pour les valeurs négatives $4{,}94\text{x}10^{-324}$ à $1{,}79\text{x}10^{308}$ pour les valeurs positives

- **Les types construits :**
  - Les COMPLEXES ;
  - Les PILES ;
  - Les FILES ;
  - Les LISTES ;
  - Les TABLEAUX à plusieurs dimensions ;

1.6.5. Quelques règles du LDA sur les variables

En LDA comme dans un langage de programmation, il existe quelques règles concernant la déclaration d'une variable et du nom même de l'algorithme :

⚠ **Le nom d'une variable ne doit jamais contenir :**
- Un espace ; ex : ~~Mon Nom~~ mais MonNom ou mon_nom
- Des caractères spéciaux ni de chiffres en début du nom de la variable ; ex : ~~6Toto, {age, l'age,~~
-

⚠ **Le nom d'une variable doit être significatif**
- Ex : AgeEtudiant ou age_etudiant (les deux syntaxes sont possibles)
- X, mauvaise nomenclature

1.7. Notions de base

1.7.1. La lecture

La lecture d'une variable permet à l'utilisateur de saisir une valeur et de la stocker dans une variable. Ici, c'est la machine qui lit la valeur que saisit l'utilisateur. D'où la notion de lecture.

En LDA on utilise l'instruction LIRE(variable1,variable2,…)

Syntaxe

```
Var Age : Entier
…
LIRE(Age)
```

Sémantique :

Dans cet exemple, l'utilisateur tapera sur le clavier la valeur d'un entier et cette valeur sera stockée dans la variable Age.

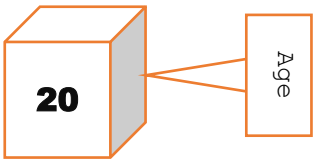


Figure 2 : Représentation d'une

1.7.2. L'écriture

L'écriture permet d'afficher un message à l'écran à destination de l'utilisateur de l'algorithme. L'ordinateur écrit (affiche) à l'écran un résultat.

En LDA on utilise l'instruction ECRIRE(expression1 ou var1,expression2 ou var2,…)

Syntaxe

```
1. ECRIRE("Texte à afficher")
2. ECRIRE( Nom," ",Prenom," ",Age)
```

Sémantique :

A la ligne 1, l'expression est une chaine de caractère qui doit apparaitre à l'écran.
A la ligne 2, ECRIRE(var1, " " ,var2," ",var3)pourrait afficher à l'écran le message suivant :

1.7.3. L'affectation

L'affectation permet d'effectuer un calcul et de stocker le résultat de ce calcul ou une donnée dans une variable.

- **Syntaxe :**

```
Nom_variable ← expression
```

- **Sémantique :**

On commence par évaluer l'expression puis on stocke le résultat dans la variable.

- Exemple :

```
Var Valeur : Entier
Valeur ← 15
Valeur ← 19
Valeur ← Valeur*3
Valeur ← 19*3
Ecrire(Valeur) # affichera 57, Pourquoi selon vous ?
```

1.7.4. Les commentaires

Les commentaires sont des explications destinées à un programmeur qui lit l'algorithme et lui permettent de mieux le comprendre. Un commentaire n'est pas exécuté(pris en compte) au cours de l'exécution d'un algorithme.

- **Différentes syntaxes :**

```
#Ici un commentaire que nous allons utiliser dans ce cours
```

Cette syntaxe sera celle que nous allons utiliser dans ce cours. En voici un autre :

```
/*une autre manière de faire un commentaire*/
```

```
/*un commentaire
Un autre sur une autre ligne
Un dernier commentaire */
```

!Ici un commentaire !

1.8. Les opérateurs

En LDA, certains opérateurs nous permettent des calculs. En voici une liste non exhaustive.

	Symboles	Signification	a b
Comparaison	<	Infériorité	a < b
	>	Supériorité	a > b
	<=	Infériorité ou égalité	a <= b
	>=	Supériorité ou égalité	a >= b
	==	Égalité	a == b
	<>	Différence	a <> b
Logique	OU	Inclusif	a OU b
	ET	Juxtaposition	a ET b
	NON	Négation	a NON b
Opération arithmétique	+	Addition	a + b
	-	Soustraction	a - b
	*	Multiplication	a * b
	/	Division euclidienne	a / b
	%	Reste de la division de deux nombres	a % b
	^	Exponentielle	a ^ b
Opération d'affectation	←	Permet d'attribuer une donnée ou un calcul à une variable	a ← b

Effectuer les calculs un a un dans l'algorigramme
Faire toujours la phase d'analyse et identifier les différents etapes au prealable

1.9. Travaux dirigés 1

Exercice 1 (contenu des variables)

Que vaut la valeur de la variable A après l'exécution de chaque bloc d'instruction ?

Cas 1 :
```
A ← 8
A ← 8 * 2
B ← A
```

Cas 2 :
```
B ← 16
A ← 9
A← A + B
```

Cas 3 :
```
A ← VRAI
B ← NON A
A ← A ET B
```

Cas 4 :
```
C ← 2017
B ← C^2
A ← A ==B
```

Cas 5 :
```
B ← 2017
A ← 2018
C ← A
```

Cas1 : A vaut :..... | Cas 2 : A vaut : ..... | Cas 3 : A vaut : ..... | Cas 4 : A vaut : .....

Case 5 : A vaut : .....

Exercice 2 (Calcul de la TVA)

1. Comment calcule-t-on le prix à payer en fonction de la quantité et du prix unitaire d'un produit sachant que le taux de la TVA (Taxe à Valeur Ajouté) des de 20% ?

Analysons le problème ensemble :

Cet exercice réclame une connaissance supplémentaire de l'étudiant à savoir la formule de calcul du prix à payer.

On a : *le prix à payer* = *prix du produit* + *le prix de la TVA*

Or *le prix de la TVA* = *Taux de la TVA x prix du produit*

Et *le prix du produit* = *prix unitaire x quantité de produit*

2. Traduire cette routine en des suite d'instructions de sorte à obtenir le résultat.

- **Identification des variables**
  - Les données d'entrées :
    - la quantité de produit Qt ;
    - le prix unitaire du produit Pu ;
    - le taux de TVA sur le produit TauxTVA ;
  - Les sorties : le prix de revient du produit Prix_de_revient

- **La spécification des résultats** (Le traitement)
  - Pour calculer le prix de revient du produit il faut d'abord connaitre la quantité du produit. Donc saisir la quantité Qt ;
  - Saisir le prix unitaire du produit Pu ;
  - Calculer le prix de revient du produit avec la formule de calcul déjà factorisé ici Prix_de_revient = Pu*Qt*(1+TauxTVA) ;
  - Afficher le résultat (c'est-à-dire le prix de revient du produit).

- **Analyse des données**

Nom de la variable	Type	Rôle / Valeur	Désignation
Qt	Réel	Donnée (DE)	La quantité des articles
Pu	Réel	Donnée (DE)	Prix unitaire
Prix_de_revient	Réel	Résultat (DS)	Prix à payer
TauxTVA	Réel	Constant (0.20)	Taux de TVA

3. Examiner ce corps d'algorithme et donner les commentaires qu'il faut

- Le corps de l'algorithme

```
ALGORITHME Calcul_TVA # Le nom de l'algorithme

# Déclaration de la constante
CONST
      TauxTVA ← 0.2 : REEL

# Déclaration des variables
VAR
      Qt, Pu, Prix_de_revient : REEL

# Le corps de l'agorithme ou le Traitement
DEBUT
      ECRIRE("Entrer la quantité : ")# on affiche à l'ordinateur
      LIRE(Qt) # l'utilisateur doit saisie une valeur qu'on stocke dans Qt
      ECRIRE("Entrer le prix unitaire : ")
      LIRE(Pu)
      Prix_de_revient ← Pu*Qt(1 + TauxTVA) # l'ordinateur effectue les calculs
      ECRIRE("Le prix est : ", Prix_de_revient)
FIN # Fin du traitement
```

Exercice 3

Ecris un algorithme qui calcule la surface et volume d'une sphère dont le rayon est introduit par l'utilisateur.

Souvenez-vous ! surface du cercle = …x Pi (avec Pi = 3.14)

Exercice 4

Ecrire un algorithme qui demande à l'utilisateur de saisir deux nombres entiers et d'inverser le contenu des deux variables.

Je vous aide un peu : les deux nombres doivent être stockés dans deux variables différentes. Le problème ici c'est de changer les valeurs. Faites attention pour ne pas écraser le contenu d'une variable. Ça sera une grande perte d'information et le résultat serait très faut ! pensez donc à une troisième variable intermédiaire….

Exercice 5

Ecrire un algorithme qui permet de saisir trois (3) nombres et d'afficher leur somme, leur différence, leur produit et leur moyenne arithmétique.

Correction

2. Les structures de contrôle

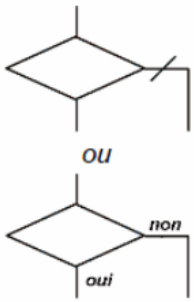
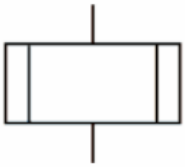
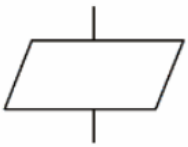
2.1. Les tests ou les structures conditionnelles

2.1.1. Les algorigrammes

Un algorigramme est la représentation graphique d'un algorithme utilisant des symboles normalisés.

En réalité c'est un diagramme qui permet de représenter et d'étudier le fonctionnement des automatismes de types séquentiels comme les chronogrammes ou le GRAFCET mais davantage réservé à la programmation des systèmes microinformatiques ainsi qu'à la maintenance.

Le diagramme est une suite de directives composées d'actions et de décisions qui doivent être exécutés selon un enchaînement strict pour réaliser une tâche (ou séquence).

SYMBOLE	DÉSIGNATION	SYMBOLE	DÉSIGNATION
	**début** ou **fin** d'un algorithme		**Test ou Branchement conditionnel** décision d'un choix parmi d'autres en fonction des conditions
	symbole général de « **traitement** » opération sur des données, instructions, … ou opération pour laquelle il n'existe aucun symbole normalisé		**sous-programme** appel d'un sous-programme
	**entrée / sortie**		**Liaison** Les différents symboles sont reliés entre eux par des lignes de liaison. Le cheminement va de haut en bas et de gauche à droite. Un cheminement différent est indiqué à l'aide d'une flèche
	**commentaire**		

2.1.2. Les tests

Un test est la vérification d'une condition logique. Le test est soit Vrai ou Faux, il est donc booléen. Il serait préférable de connaitre les différentes opérations ou conditions possible.

Exemples :

- **Test 1** : (4 < 2) # Le Test 1 est Faux
- **Test 2** : (8 == 4*2) # Le Test 2 est Vrai
- **Test 3** : (5 > 6) ET (8 < 12) # Le Test 3 est Vrai

Nous vous proposons les tables de vérité ici :

ET	VRAI	FAUX
VRAI	Vrai	Faux
FAUX	Faux	Faux

OU	VRAI	FAUX
VRAI	Vrai	Vrai
FAUX	Vrai	Faux

2.1.3. Les structures conditionnelles simples

Cette structure permet de traiter un bloc d'instruction quand le test est vrai, Dans le cas où le test est faux, la structure ne sera pas exécutée.

Syntaxe 1

```
SI (Test)ALORS
DEBUT
        Instructions #DEBUT et FIN s'il y a plusieurs instructions à exécuter
FIN
```

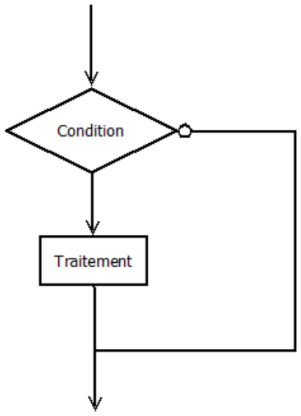
Syntaxe 2

```
SI (Test)ALORS
        Instructions # Omission de DEBUT et Ajout de FINSI
FINSI
```

Syntaxe 3

```
SI (Test)ALORS
        Instruction # Omission de DEBUT et de FIN car une seule instruction à
                    #exécuter Facultatif dans ce cas
```

Algorigramme



Exercice 6

Ecrire un algorithme qui permet de vérifier si un utilisateur est adulte c'est-à-dire que son âge est supérieur ou égal à 18.

Exercice 6 (correction)

- Nous avons besoin de saisir d'abord l'âge de l'individu
- Tester pour voir si cet individu est adulte.

```
Algorithme Adulte
Var
        Age : Entier
Debut
        Ecrire( "Saisissez votre âge SVP : ")
        Lire(Age)
        Si(Age>18)Alors
                Ecrire("Vous êtes bien un adulte ")
        FinSi
Fin
```

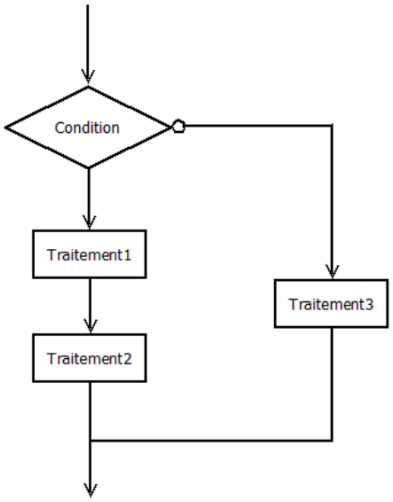
2.1.4. Les structures conditionnelles alternatives

Cette structure permet de traiter un bloc d'instruction quand le test est vrai ou un autre bloc si le test est faux. Dans les deux cas l'un des deux blocs sera exécuté.

Syntaxe

```
SI (Test)ALORS
        Instructions 1 #DEBUT et FIN s'il y a plusieurs instructions à
        exécuter
SINON
        Instructions 2
FINSI
```

Algorigramme



Remarque : dès que le SI est vrai, automatique le SINON est faux et vice versa. A l'exécution on choisira le branchement dont le test donne vrai. Un seul branchement est possible.

Modifier l'algorithme de l'exercice 6 de sorte à afficher la phrase « vous êtes mineur » lorsque l'âge de la personne est inférieur à 18.

Exercice 7

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on laisse de côté le cas où le nombre vaut zéro).

Exercice 7 (correction)

```
Algorithme SigneNombre
Var
        a : Reel
Debut
        Ecrire("Entrer le nombre svp ")
        Lire(a)
        Si (a<0) Alors
                Ecrire("Le nombre est négatif")
        Sinon
                Ecrire("Le nombre est positif ou nul")
        FinSi
Fin
```

2.1.5. Les structures conditionnelles imbriquées

Cette structure est l'emboitement de plusieurs structures conditionnelles alternatives. Elle permet de passer de deux possibilités à plusieurs.

Syntaxe

```
SI Test 1 ALORS
        Instructions
SION
        SI (Test 2)Alors
                Instructions
        FiNSI
FINSI
```

Analysons l'algorithme suivant

Un algorithme qui donne l'état (liquide, gazeux ou solide) de l'eau en fonction de la température donnée par un utilisateur.

```
Algorithme TemperatureEau
Var
        Temp : Reel
Debut
        Ecrire("Enrer la température svp : ")
        Lire(Temp)
        Si(Temp <= 0)Alors
                Ecrire("L'eau est à l'état solide")
        FinSi
        Si(Temp > 0 ET Temp < 100)Alors
                Ecrire("L'eau est à l'état liquide")
        FinSi
        Si(Temp > 100)Alors
                Ecrire("L'eau est à l'état gazeux")
        FinSi
Fin
```

Il n'est pas possible d'avoir à la fois de l'eau à l'état solide et à l'état liquide en même temps, la température étant unique. Si la première condition est vraie alors les autres conditions sont fausses. Mais la machine testera quand même les autres conditions. Quelle perte de temps ! Pour pallier ce problème, il faut donc réduire ces trois conditions en deux conditions alternatives : on parle alors d'imbrication de condition.

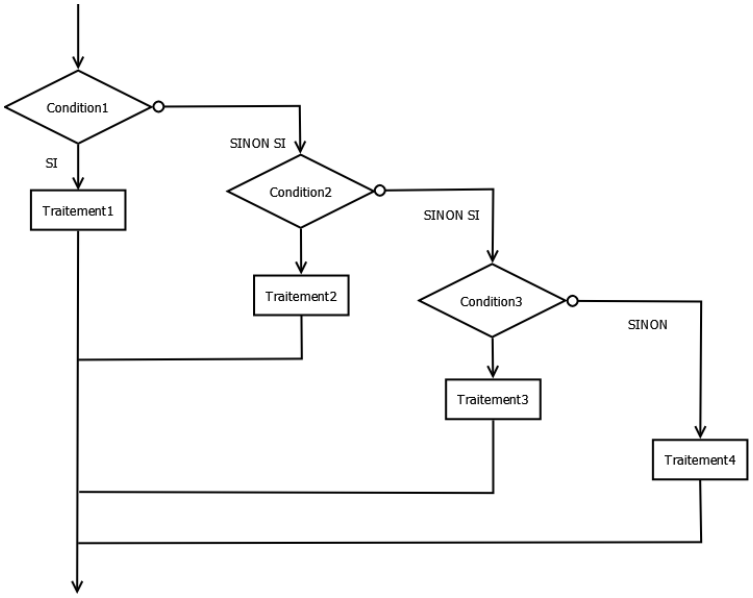
```
Algorithme TemperatureEau
Var
        Temp : Reel
Debut
        Ecrire("Entrer la température svp : ")
        Lire(Temp)
        Si(Temp <=0)Alors
                Ecrire("L'eau est à l'état solide")
        SiNon
                Si(Temp <100)Alors
                        Ecrire("L'eau est à l'état liquide")
                SiNon
                        Ecrire("L'eau est à l'état gazeux")
                FinSi
        FinSI
Fin
```

Ainsi, dans tout le bloc de Si ...SiNon FinSI Si ... SiNon FinSi un seul branchement sera possible.

Nous avons fait des économies : au lieu de devoir taper trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur **le temps d'exécution de l'ordinateur**. Si la température est inférieure à zéro, celui-ci écrit dorénavant « C'est de la glace » et passe directement à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).
Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.

Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

Algorigramme



2.1. Travaux dirigés 2

Exercices

Exercice 8

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit pas calculer le produit des deux nombres.

Exercice 9

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on exclut cette fois le traitement du cas où le nombre vaut zéro).

Exercice 10

Ecrire un algorithme qui demande trois nombres à l'utilisateur et l'informe ensuite s'ils sont rangés ou non dans l'ordre croissant.

Exercice 11

Ecrire un algorithme qui demande un nombre à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif (on inclut cette fois le traitement du cas où le nombre vaut zéro).

Exercice 12

Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si le produit est négatif ou positif (on inclut cette fois le traitement du cas où le produit peut être nul). Attention toutefois, on ne doit pas calculer le produit !

Exercice 13

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- "Poussin" de 6 à 7 ans
- "Pupille" de 8 à 9 ans
- "Minime" de 10 à 11 ans
- "Cadet" après 12 ans

Peut-on concevoir plusieurs algorithmes équivalents menant à ce résultat ?

Corrections

2.2. Les boucles ou structures itératives

Grâce aux structures itératives, on a la possibilité d'emmener l'ordinateur à exécuter plusieurs fois une séquence d'instructions jusqu'à une condition d'arrêt donnée. Il existe une variété de structures itératives en fonction du besoin.

2.2.1. La boucle Pour … Faire … FinPour

Cette boucle exécute un bloc d'instruction un certain ***nombre finie*** de fois définie par l'utilisateur. Elle se caractérise par le fait que l'on connait à l'avance le nombre d'itérations que l'on va devoir effectuer.

- A chaque instant, on connait le nombre d'itérations déjà effectuées.
- On connait aussi le nombre d'itérations restantes.

➡ **Syntaxe**

Pour *(variable)* **allant de** *(début)* **A** *(fin)* **par** *(pas)* **Faire**
    Instructions
Fin Pour

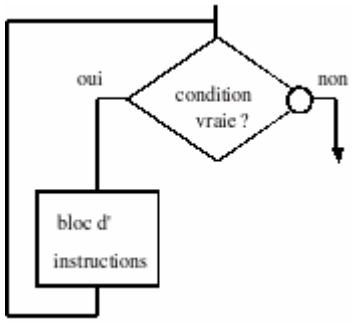
Pour *variable* ← *debut A fin, pas* **Faire**
    Bloc d'instructions
FinPour

➡ **Exemple :** table de multiplication de 8

```
Algorithme Multiplication_8
Var
        i :Entier
Debut
        Pour i ← 1 A 10 Faire #Lorsque le pas n'est pas spécifié, il vaut 1
                Ecrire(i, "x 8  = ",i*8)
        FinPour
Fin
```

➔ Algorigramme :

Ecrire l'algorigramme de cette boucle POUR !



2.2.2. La boucles TantQue… Faire … FInTanQue

Les boucles TantQue permettent d'effectuer des itérations tant qu'une certaine condition est vérifiée. On ne connait pas le nombre d'itérations à effectuer, mais à chaque itération, on vérifie si la condition est vraie ou fausse. Dès que cette condition est fausse, on sort de la boucle. Son nombre minimum d'exécution est de 0 fois.

On sait qu'à la sortie de la boucle, la condition de boucle est fausse.

⚠ Attention : il faut s'assurer que les itérations permettent de modifier la valeur de la condition de boucle, si ce n'est pas le cas, la boucle ne s'arrête jamais. On parle alors de **boucle infinie**.

Un exemple : si on effectue une boucle Tantque dont la condition de poursuite est (a==0), si la variable a n'a pas de chance d'être modifiée dans la boucle, il ne sera pas possible de sortir de cette boucle

➔ Syntaxe

```
TANTQUE (Test) FAIRE
   instruction1
          ……...
   InstructionN
FINTANTQUE
```

⇔

```
TantQue (Test) Faire
  instruction1
                ……...
  InstructionN
FinTantQue
```

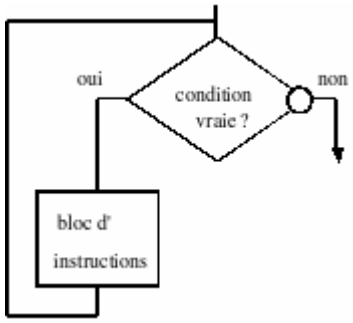
On peut avoir également la syntaxe suivante :

```
TantQue (conditions) Faire
      Debut
            instruction1
       ……...
            InstructionN
      Fin
```

➔ **Exemple** (Tester cet algorithme pour a = 3 et b = 14)

```
Algorithme DivisionEntiere
Var
      a,b : Entier
Debut
      Lire(a,b)
      TantQue (a<b) Faire
            b ← b-a
            Si(a>=b)Alors
                  Ecrire("La division entière est :",b)
            FinSi
      FinTantQue
Fin
```

➔ Algorigramme



2.2.3. La boucles Repeter…JusquA

La boucle REPETER permet également à la machine d'exécuter un certain nombre d'instruction jusqu'à ce qu'une ou plusieurs condition(s) soient satisfait(es). De ce fait, cette boucle s'exécute au minimum une fois.

➔ Syntaxe

```
REPETER
      Instruction1
      …
      intructionN

JUSQUA(condition(s) vrai)
```

⇔

```
Repeter
      Instruction1
      …
      intructionN

JusquA(condition(s) vrai)
```

➔ **Exemple** : un algorithme qui dit si un nombre est premier ou ne l'est pas !

Quand dit-on qu'un nombre est premier ?

Rappelez-vous de la définition de la classe de 5ième
Ou de celle de la classe de 3ième

Un nombre est premier lorsqu'il n'admet pas (en dehors de 1 et lui-même) de diviseurs dans la liste de tous les nombres positifs consécutifs inférieurs à lui.

```
Algorithme NombrePremier
Var
        a,i : Entier
        Test : Booleen
Debut
        Test ← Vrai #C1
        Lire(a)
        i ← 2 #C2
        Repeter
                Si(a % i == 0)Alors #C3
                        Ecrire("Le nombre ",a, "n'est pas premier")
                        Test ← Faux #C4
                SiNon
                        i ← i+1 #C5
                FinSi
        JusquA(i==a OU Test==Faux) #C6
        Si(i==a)Alors #C7
                Ecrire("Le nombre ",a,"est premier")
        FinSi
Fin
```

Commentez les lignes suivantes :

#C1 :……

#C2 :……

#C3 :……

#C4 :……

#C5 :……

#C6 :……

#C7 :……

- Cet algorithme est prouve-t-il que 2 est un nombre premier ?
- Proposer un autre algorithme qui prend en compte le fait que le nombre 2 est premier

2.3. Travaux dirigés 3

Exercices

Exercice 14

En utilisant l'exemple ci-avant, écrivez un algorithme qui affiche les nombres premiers inferieurs à N = 10 000.

Exercice 15

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

Exercice 16

Ecrire un algorithme qui demande à l'utilisateur un nombre compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 17

Ecrire un algorithme (sans utiliser la boucle **Pour**) qui demande un nombre de départ, et qui ensuite affiche les dix nombres suivants. Par exemple, si l'utilisateur entre le nombre 17, le programme affichera les nombres : 18 19 20 21 22 23 24 25 26 27.

Exercice 18

Réécrire l'algorithme précédent, en utilisant cette fois l'instruction **Pour**

Exercice 19

Ecrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :

```
Table de 7 :
7 x 1 = 7
7 x 2 = 14
```

...
7 x 10 = 70

Exercice 20

Ecrire un algorithme qui demande un nombre de départ, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre 5, le programme doit calculer :

$1 + 2 + 3 + 4 + 5 = 15$

NB : on souhaite afficher uniquement le résultat, pas la décomposition du calcul.

Exercice 21

Ecrire un algorithme qui demande un nombre de départ, et qui calcule sa factorielle.
NB : la factorielle de 8, notée 8 ! vaut 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8

Exercice 22

Ecrire un algorithme qui demande successivement 20 nombres à l'utilisateur, et qui lui dit ensuite quel était le plus grand parmi ces 20 nombres :

Entrez le nombre numéro 1 : 12
Entrez le nombre numéro 2 : 14
etc.
Entrez le nombre numéro 20 : 6
Le plus grand de ces nombres est : 14

Modifiez ensuite l'algorithme pour que le programme affiche de surcroît en quelle position avait été saisie ce nombre :

C'était le nombre numéro 2

Exercice 23

Réécrire l'algorithme précédent, mais cette fois-ci on ne connaît pas d'avance combien l'utilisateur souhaite saisir de nombres. La saisie des nombres s'arrête lorsque l'utilisateur entre un zéro.

Exercice 24

Quelles différences faites-vous entre les boucles Pour, TantQue et Repeter ?
Le nombre d'itération est le nombre maximal de fois que la boucle peut s'exécuter : ce nombre peut être défini, ou indéfini d'avance.

BOUCLE	Nombre max d'itération	Exécution minimale
POUR	……	……
TANTQUE	……	……
REPETER	……	……

Correction

3. Les sous-algorithmes

3.1. Contexte

Une application, surtout si son code source est long, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, si l'on veut faire un contrôle de saisie sur plusieurs champs d'entrées, on aura presque besoin du même code à chaque fois.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie les lignes de codes voulues en ne changeant que le nécessaire, et roule Raoul. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent...

- D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, il peut devenir parfaitement illisible. Or, le fait d'être facilement modifiable donc lisible, y compris - et surtout - par ceux qui ne l'ont pas écrit est un critère essentiel pour un programme informatique ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière aiguë. L'ignorer, c'est donc forcément grave.

- En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification ! Et si l'on en oublie une, patatras, on a laissé un bug.

- Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à regrouper les instructions qui le composent en un module séparé. Il ne restera alors plus qu'à appeler ce groupe d'instructions (qui n'existe donc désormais qu'en un exemplaire unique) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient modulaire, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors la **procédure principale**, et ces groupes d'instructions auxquels on a recours s'appellent des **fonctions** et des **sous-procédures** (nous verrons un peu plus loin la différence entre ces deux termes).

Un sous-algorithme est en lui-même un algorithme. De ce fait il bénéficie de la même structuration qu'un algorithme que nous avons connu depuis lors.

3.2. Les sous-procédures

3.2.1. Définition

Une sous-procédure que nous appelons une procédure est un algorithme qui accomplit une tâche bien spécifique. Il est conçu pour être réutilisé à plusieurs reprises au sein d'une procédure principale.

Syntaxe 1

```
Procedure NomProcedure()
Const
#Déclaration des constantes
Var
      #déclaration des variables
Debut
      #Suites d'instructions
FinProcedure
```

Illustration

Exercice 25 :

Ecrire une procédure *SaisieNom()* qui permet à un utilisateur de pouvoir saisir son nom.

```
Procedure SaisieNom(paramères facultatifs)
Var
      Nom : Chaine
Debut
      Ecrire("Veillez entrer votre nom SVP :")
      Lire(Nom)
FinProcedure
```

⚠ Il n'est pas conseillé d'oublier les parenthèses ouvrante et fermante devant le nom de la procédure.

Cette procédure SaisieNom() est appelée une procédure sans paramètre.

➔ Syntaxe 2

```
Procedure NomProcedureParametre(para1 :Type1, para2, para3 :Type2,…)
Const
#Déclaration des constantes
Var
       #déclaration des variables
Debut
       #Suites d'instructions
FinProcedure
```

Un paramètre est une variable déclarée entre les deux parenthèses de la procédure. Cette variable pourra être utilisée par la procédure en question.

➔ Appel

Appeler une procédure c'est de lui faire référence, c'est de l'utiliser.
La syntaxe pour cela est :

```
Debut
       SaisieNom()
Fin
```

3.3. Les fonctions

3.3.1. Les fonctions prédéfinies

Certaines fonctions sont déjà connues et programmées par des programmeurs depuis plusieurs années. Il suffit de les connaître afin de leur faire appel si besoin s'en suit.

Exemple : Sin( ), Cos( ), Abs( ), Tan( ), Exp( ), Log( ), Sqrt( )…

Dans un algorithme en LDA on les utilise selon la syntaxe suivante :

```
a ← Sin(1.2)
b ← Tan(cos(6.25))
c ← Log(45)/Sqrt(20)
```

3.3.2. Les fonctions personnalisées

Il arrive parfois dans la programmation que l'on veille écrire soi-même une fonction inexistante. Cela est possible. Il suffit de juste savoir qu'une fonction est une sous-procédure qui renvoie obligatoirement une valeur unique à son utilisateur. Elle s'utilise toujours avec des paramètres.

Syntaxe

```
Fonction NomFonctions(para1 :Type, para2, para3 :Type,…)Type renvoyé
Const
       #Déclaration des constantes
Var
       #déclaration des variables
Debut
       #Traitements
Retourner Résultat
```

Le résultat renvoyé (Résultat) par la fonction doit être du type que Type renvoyé

Si vous remarquez bien, une fonction n'a pas l'instruction *FinFonction* mais **Retourner.** Cela est ainsi parce que la fonction, une fois la valeur renvoyée se dit avoir terminée sa tâche.

Exercice 26 :

Ecrire une sous-procédure ou fonction qui calcule et renvoie le carré d'un nombre réel.

```
Fonction squart(a :Reel):Reel
Var
                    Carre : Reel
Debut
                    Carre ← a*a
Retourner Carre
```

```
Fonction squart(a :Reel):Reel
      Retourner a*a
```

➔ Appel

Pour faire appel à une fonction c'est très facile. Comme une fonction renvoie toujours une valeur, il serait souvent convenable de stocker cette valeur retournée dans une variable.

Exemple :

```
a ← Squart(4)# a vaut 16
```

3.4. La procédure principale

➔ Syntaxe

La procédure principale que nous avons appelé depuis le début de ce cours Algorithme est en générale la composition de sous procédures et de fonctions. La question que nous nous posons est comment donc devons-nous disposer ces sous procédures par rapport à la procédure principale ?
Les sous procédures doivent être déclarées au sein de la procédure principale : soit en début après la déclaration des variables ou à la fin de l'algorithme avant le mot **Fin**.
En réalité, une sous procédure peut être déclarée n'importe où dans la procédure principale mais cela reste très problématique pour le programmeur. Je vous conseille donc d'utiliser les syntaxes suivantes.

```
Algorithme NomAlgo
Const #déclaration des constantes
Var
      #déclaration des variables
#déclaration des sous-procédures ou fonctions
Debut
      Traitements
Fin
```

Ou encore :

```
Algorithme NomAlgo
Const #déclaration des constantes
Var
     #déclaration des variables
Debut
     Traitements
#déclaration des sous-procédures ou fonctions
Fin
```

3.4.1. La portée des variables

Visibilité

Vue que les sous procédures sont déclarées au sein de la procédure principale, nous constatons ensemble qu'il y aura plusieurs déclarations de variable. Notamment les variables déclarées dans la procédure principale et les variables déclarées dans les sous procédures.

Supposons qu'une variable *Toto* soit déclaré au sein d'une sous procédure et que nous voulons accéder à celle-ci au sein de la procédure principale. Cela serait-il possible ?

- Les variables déclarées au sein de la procédure principale sont **visibles** par toutes les sous procédures de ladite procédure principale. Visibles voudrait dire que leurs contenus sont modifiables par les sous procédures. Elles sont donc accessibles partout dans la procédure principale. Ces variables sont appelées les **variables globales**.
- Mais pour **les variables locales**, celles qui sont déclarées au sein d'une sous procédure ne sont visibles que par la sous procédure elle-seule. A l'extérieur c'est-à-dire dans la procédure principale ou dans une autre sous procédure ces variables ne sont pas reconnues.

Illustration

Exercice 27 :

Ecrire un algorithme qui demande à son utilisateur d'entrer un entier compris entre 20 et 50, puis dit à ce dernier si ce nombre est plus proche de 20 ou de 50. On dit qu'un nombre x est proche de 50 lorsque |x-50|< |x-20|.

Analyse du problème

Etape 0 : la compréhension du problème : il s'agit de prendre un nombre x et de le comparer avec les deux valeurs 20 et 50 puis de calculer la distance de ce nombre à ces deux valeurs.

Etape 1 : La nomenclature : PlusProche
Etape 2 : identification des informations :

- **Les données :** les valeurs 20 et 50, le nombre Nb fournit par l'utilisateur
- **Les intermédiaires :** d20 : la distant de Nb à 20 et d50 la distance de 50 à Nb, TestNb : pour tester si Nb est dans l'intervalle indiqué. Rep : stocke l'une des deux phrases ci-dessous.
- **Les résultats ou les sorties :** phrase « Plus proche de 20 » ou « Plus proche de 50 » que nous stockons dans les variables Phrase1 et Phrase2

Etape 3 : l'idée générale : ce problème est un calcul de distance en vue de ramener la plus petite distance d'un point distinct à deux bornes déjà connues.

Etape 4 : Les principales étapes (Traitement)

1- Saisir le nombre Nb correcte fournir par l'utilisateur
2- Vérifier si ce nombre est valide c'est-à-dire compris entre 20 et 50
3- Calculer les deux distances d20 et d50
4- Donner le résultat

Etape 5 : dresser la liste des variables, constantes et types

Nom Variable	Type	Rôle/Valeur	Désignation
Binf	Const Entier	20	Borne inférieure
Bsup	Const Entier	50	Borne supérieure
Nb	Entier	Donnée	Nombre saisie par l'utilisateur
d20	Entier	Intermédiaire	Distance de Nb à 20
x	Entier	Paramètre	Variable temporelle (à voir)
d50	Entier	Intermédiaire	Distance de Nb à 50
TestNb	Booleen	Intermédiaire	Pour teste si Nb ε [20,50]
Phrase1	Chaine	Sortie	" Plus proche de 20 " Nb plus proche de 20
Phrase2	Chaine	Sortie	" Plus proche de 50 " Nb plus proche de 50

Rep	Chaine	Intermédiaire	Stocke Phrase1 ou Phrase2

Etape 6 : les procédures et fonctions.

» Ecriture du corps de la procédure qui permet de saisir le nombre Nb correctement fournit par l'utilisateur. Cette sous procédure va nous dire si l'utilisateur a saisi la bonne valeur Nb ou pas. Il nous faut donc une variable de type booléen (deux états : Vrai ou Faux). Vrai « Nb correct » et Faux « Nb en dehors de l'intervalle donné ».

```
Procedure ControleSaisie(x : Entier)
Debut
      Si(x<50 ET x >20)Alors
            TestNb ← Vrai
      SiNon
            TestNb ← Faux
      FinSi
FinProcedure
```

» Ecriture de la fonction qui calcule la distance de Nb à d20 et à d50.

Cette fonction que nous appelons *CalculDistance()* va renvoyer un entier 20 ou 50 parce que dans ce problème nous traitons des entiers. Il nous faut une valeur absolue afin de respecter le type Entier positif (distance).

```
Fonction CalculDistance (x :Entier):Entier
Var
      d20,d50 : Entier
Debut
      d20 ← x - 20
      d50 ← 50 - x
      Si(d20 < d50)Alors
            Retourner 20
      SiNon
            Retourner 50
      FinSi
Fin
```

Etape 7 : l'algorithme

Exercice 27 (correction)

```
Algorithme PlusProche
Var
      TestNb : Booleen
      Nb : Entier
      Rep : Chaine

#déclaration des sous-procédures ou fonctions

Procedure ControleSaisie(x : Entier)
Const
      Binf ← 20, Bsup ← 50 : Entier
Debut
      Si(x<Bsup ET x >Binf)Alors
            TestNb ← Vrai
      SiNon
            TestNb ← Faux
      FinSi
FinProcedure

Fonction CalculDistance (x :Entier):Chaine
Var
      d20, d50 : Entier
      Phrase1, Phrase2 : Chaine
Debut
      Phrase1 ← "Plus proche de 20"
      Phrase2 ← "Plus proche de 50"
      d20 ← x - 20
      d50 ← 50 - x
      Si(d20 < d50)Alors
            Retourner Phrase1
      SiNon
            Retourner Phrase2
      FinSi
Fin
#Début de la procédure principale

Debut
      Repeter
            Ecrire("Entrez un entier compris entre 20 est 50 SVP ")
            Lire(Nb)
            ControleSaisie(Nb)#Appel de la sous procédure
            Si (TestNb == FAUX)Alors
                  Ecrire ("Entrée incorrecte ! Ressayer ")
      JusquA(TestNb == Vrai)
      Rep ← CalculDistance(Nb) )#Appel de la fonction avec Nb correcte
      Ecrire(rep)
Fin
```

Analyse de l'algorithme

- Les variables TestNb et Nb sont des variables globales. Elles sont sollicitées par la sous procédure *ControleSaisie()* en vue de modifier leurs contenus. Elles sont visibles et donc accessibles partout dans l'algorithme.
- Les variables Phrase1, Phrase2, d20, d50 sont des variables locales. Après leur utilisation par la fonction *CalculeDistance*(), elles sont automatiquement détruites. C'est la raison pour laquelle elles ne sont pas visibles par la procédure principale.

La variable **x** est variable passée en paramètre !!! Nous allons l'étudier.

3.4.2. Le mode de passage des variables

Dans le cas de la variable x, il existe deux types de passage. On appelle la variable x un argument. L'argument peut être passé par *valeur* ou par *référence*.
Voyons de près de quoi il s'agit :

Le passage par valeur

Pour qu'un argument soit passé par valeur il faut qu'il soit une variable globale. La sous procédure crée une copie de cette variable et travaille sur cette nouvelle copie. Ce qui laisse l'original intact, sans changement de valeur. La procédure n'influence dont pas sur cette variable passé en argument par valeur.

Aussi un paramètre passé par valeur ne peut être paramètre en entrée. C'est certes une limite, mais c'est d'abord et avant tout une sécurité : quand on transmet un paramètre par valeur, on est sûr et certain que même en cas de bug dans la sous-procédure, la valeur de la variable transmise ne sera jamais modifiée par erreur (c'est-à-dire écrasée) dans le programme principal

Le passage par référence

Passer un paramètre par référence, permet d'utiliser ce paramètre tant en lecture (en entrée) qu'en écriture (en sortie), puisque toute modification de la valeur du paramètre

aura pour effet de modifier la variable correspondante dans la procédure appelante. Mais on comprend à quel point on ouvre ainsi la porte à des catastrophes : si la sous-procédure fait une bêtise avec sa variable, elle fait en réalité une bêtise avec la variable du programme principal... on palie ce problème plus aisément en créant une copie de cette variable dans le programme principal.

Mais nous n'allons pas tout de même écrire des sous procédures qui « buguent ».

Récapitulons

	Passage par valeur	Passage par référence
Utilisation en Entrée	OUI	OUI
Utilisation en Sortie	NON	OUI

Dans notre exemple *PlusProche*, la variable x est passé par référence. C'est le cas dans le langage Python. Nous adopterons le passage par référence dans ce cours.

3.5. Travaux dirigés 4

Exercices

Exercice 28

Écrivez une fonction qui renvoie la somme de cinq nombres fournis en argument.

Exercice 29

Ecrire un traitement qui inverse le contenu de deux réels passés en argument.

Exercice 30

Ecrire un traitement qui permet de calculer la valeur absolue d'un nombre réel.

Exercice 31

Ecrire une fonction qui permet de savoir si un entier est divisible par un autre. On pourra utiliser un nouveau type nommé logique afin de renvoyer le résultat

Exercice 32

Créer un petit ensemble de procédures et de fonctions permettant de manipuler facilement les heures et les minutes et composé de :

a) La fonction Minutes, qui calcule le nombre des minutes correspondant à un nombre d'heures et un nombre de minutes donnés.

b) La fonction ou la procédure HeuresMinutes qui réalise la transformation inverse de la fonction Minute. Pour HeuresMinutes, il y a deux résultats à fournir, une fonction ne peut convenir, il faut donc écrire une procédure comportant trois paramètres : La durée (entrée), l'heure (sortie) et les minutes (sortie).

c) La procédure AjouteTemps qui additionne deux couples de données heures et minutes en utilisant les deux fonctions précédentes. La procédure AjouteTemps reçoit quatre paramètres en entrée, fournit deux paramètres en sortie. La variable locale MinuteEnTout sert à stocker un résultat intermédiaire, mais elle n'est pas indispensable.

Exercice 33

Cet exercice permet de compléter les procédures et fonctions de l'exercice précédent

1. Créer une fonction qui permet de dire si un mois a 30 jours ou non. Cette fonction renverra 1 si c'est le cas et 0 sinon.

1	2	3	4	5	6	7	8	9	10	11	12
Jan	Fev	Mars	Avril	Mai	Juin	Juil	Aout	Sept	Oct	Nov	Dec
31	28/ 29	31	30	31	30	31	31	30	31	30	31

2. Créer une fonction qui permet de dire si un mois a 31 jours ou non. Cette fonction renverra 1 si c'est le cas et 0 sinon.

3. Créer une fonction qui permet de dire si une année est bissextile ou non. Cette fonction renverra 1 si c'est le cas et 0 sinon. Pour qu'une année soit bissextile, il suffit que l'année soit un nombre divisible par 4 et non divisible par 100, ou alors qu'elle soit divisible par 400.

4. Utiliser ces fonctions pour écrire une fonction NombreDeJour()retournant le nombre de jours pour un mois et une année donnée.

5. Écrire un programme principal permettant à l'utilisateur d'entrer un numéro de mois (entre 1 et 12) et une année (entre 1995 et 2100), qui seront ensuite passés en paramètres à la fonction NombreDeJour(). Il faut tester la validité des mois et années.

Corrections

3.6. La récursivité

3.6.1. Définition

Il arrive parfois que lors de l'exécution d'une fonction, elle fait appel à elle-même. On parle alors de récursivité.
En mathématique on parle de récurrence au niveau des suites numériques ou des suites de fonction !
Exemple : f(n) = f(n-1) + f(n-2) avec f(0) = 0 , f(1) = 1 et n > 1

Dans cet exemple il s'agit de la suite de Fibonacci. Pour calculer f(n) il faut faire appel à f(n-1) et f(n-2). Une pile se forme donc au niveau des résultats parce qu'un résultat attend l'autre pour pouvoir démarrer.

3.6.2. Avantages

- La programmation récursive, pour traiter certains problèmes, est très économique pour le programmeur ; elle permet de faire les choses correctement, en très peu d'instructions.

- En revanche, elle est très dispendieuse de ressources machine. Car à l'exécution, la machine va être obligée de créer autant de variables temporaires que de « tours » de fonction en attente.

- Last but not least, et c'est le gag final, tout problème formulé en termes récursifs peut également être formulé en termes itératifs ! Donc, si la programmation récursive peut faciliter la vie du programmeur, elle n'est jamais indispensable. Mais ça me faisait tant plaisir de vous en parler que je n'ai pas pu résister… Et puis, accessoirement, même si on ne s'en sert pas, en tant qu'informaticien, il faut connaître cette technique sur laquelle on peut toujours tomber un jour ou l'autre.

3.6.3. Quelques exemples

La fonction Pgcd()

Pour calculer le Plus Grand Commun Diviseur (PGCD), nous pouvons utiliser les formules mathématiques suivantes :

a et b étant des Entiers positifs on a : si a > b alors **Pgcd**(a,b) = **Pgcd**(a-b, b) et **Pgcd**(0,a) = a. nous constatons que le calcule d'un Pgcd appel un autre calcul de Pgcd ainsi de suite jusqu'à atteindre une condition limite Pgcd(a,0) = a. c'est la récursivité. Pas de sorcellerie par ici !!!

Deux approches sont possibles : procédure ou fonction

Approche procédurale

```
Algorithme Pgcd
Var
       a, b : Entier
Debut
       Ecrire("Entrez deux nombres entiers svp ! ")
       Lire(a,b)
       TantQue(b<>a) Faire
              Si(a > b)Alors
                     a ← a - b
              SiNon
                     b ← b - a
              FinSi
       FinTantQue
       Ecrire("Pgcd =",b)
Fin
```

Approche par fonction récursive

```
Fonction Pgcd(a,b :Entier) :Entier
Debut
       Si(a==b)Alors
              Retourner a #Condition d'arrêt !
       SiNon
              Si(a>b)Alors
                     Retourner Pgcd(a-b,b)# la fonction fait appelle à elle-même
              SiNon
                     Retourner Pgcd(a,b-a)
              FinSi
       FinSi
Fin
```

Testons cette fonction avec a = 42 et b = 25

1. 42 > 25 donc on calcule Pgcd(17,25) avec a = 17 et b = 25
2. 17 < 25 donc on calcule Pgcd(17, 8) avec a = 17 et b = 8
3. 17 > 8 donc on calcule Pgcd(9,8) avec a = 9 et b = 8
4. 9 > 8 donc on calcule Pgcd(1,8) avec a = 1 et b = 8
5. 1 < 8 donc on calcule Pgcd(1,7) avec a = 1 et b = 7
6. 1 < 7 on calcule Pgcd(1,6) avec a = 1 et b = 6
7. 1 < 6 on calcule Pgcd(1,5) avec a = 1 et b = 5
8. 1 < 5 on calcule Pgcd(1,4) avec a = 1 et b = 4
9. 1 < 4 on calcule Pgcd(1,3) avec a = 1 et b = 3
10. 1 < 3 on calcule Pgcd(1,2) avec a = 1 et b = 2
11. 1 < 2 **on calcule Pgcd(1,1) avec** a = b = 1 condition d'arrêt
12. *1 = 1 (cas où a=b) donc on retourne a qui vaut 1 et la fonction arrête son travail.*

La suite de Fibonacci

Approche procédurale

```
Algorithme Fibonacci
Var
       F1,F2,F3,n,i : Entier
Debut
       F1 ← 0
       F2 ← 1
       Ecrire("Entrer la valeur de n : ")
       Lire(n)
       Pour i ← 2 A n Faire
              F3 ← F2 + F1
              F1 ← F2 # on garde les valeurs précédentes
              F2 ← F3 # on garde les valeurs précédentes
       FinPour
       Ecrire("Le ",n,"ième terme de la suite de Fibo est : ",F3)
Fin
```

Approche par fonction récursive

```
Fonction Fibo(n :Entier) :Entier
Debut
       Si(n == 0)Alors
              Retourner 0
       SiNon
              Si(n == 1)Alors
                     Retourner 1
              SiNon
                     Retourner Fibo(n-1) + Fibo(n-2)
              FinSi
       FinSi
Fin
```

Testons cette fonction Fibo pour n = 5

1. n = 5 <> 1 et n <> 0 donc calculer Fibo(5) revient à calculer Fibo(4) et Fibo(3) et faire leur somme
2. a) Fibo(4) calcule Fibo(3) et Fibo(2)
   b) Fibo(3) clcule Fibo(2) et Fibo(1)
3. Fibo(2) calcule Fibo(1) et Fibo(0)
4. Fibo(1) Retourne 1 et Fibo(0) Retourne 0
5. Puis on remonte en faisant la somme et on obtient le résultat.

Ces tableaux se lisent du bas vers le haut.

Fibo(n)	Fibo(n-1)	Fibo(n-2)	Somme
Fibo(5)	Fibo(4)	Fibo(3)	
Fibo(4)	Fibo(3)	Fibo(2)	
Fibo(3)	Fibo(2)	Fibo(1)	
Fibo(2)	Fibo(1)	Fibo(0)	1
Fibo(1)	1		
Fibo(0)	0		

Fibo(n)	Fibo(n-1)	Fibo(n-2)	Somme
...	...	...	...
Fibo(7)	8	5	13
Fibo(6)	5	3	8
Fibo(5)	3	2	5
Fibo(4)	2	1	3
Fibo(3)	1	1	2
Fibo(2)	1	0	1
Fibo(1)	1		
Fibo(0)	0		

Somme()

Écrire une fonction récursive qui calcule la somme de nombres de 1 à n, si n > 0 et renvoie 0 sinon

Approche par fonction récursive

```
Fonction Somme(n :Entier) :Entier
Debut
       Si(n > 0)Alors
              Retourner (Somme(n-1) + n)
       SiNon
              Retourner 0
       FinSi
Fin
```

Testons cette fonction Somme(5) (pour n = 5)

- Somm(5) = Somme(4) + 5
- Somm(5) = Somme(3) + 9
- Somm(5) = Somme(2) + 12
- Somm(5) = Somme(1) + 14
- Somm(5) = Somme(0) + 15
- Somm(5) = 0 + 15

Donc Somme(15) = 15

Le mécanisme qui se passe !

☞ Somm(5) = Somme(4) + 5
☞ Somm(5) = Somme(3) + 4 + 5
☞ Somm(5) = Somme(2) + 3 + 4 + 5
☞ Somm(5) = Somme(1) + 2 + 3 + 4 + 5
☞ Somm(5) = Somme(0) + 1 + 2 + 3 + 4 + 5
☞ Somm(5) = 0 + 1 + 2 + 3 + 4 + 5

4. Les types construits

4.1. Les tableaux

4.1.1. Définition

Un ensemble de valeurs portant le même nom de variable et repérées par un nombre, s'appelle un tableau, ou encore une variable indicée.

Tab :

45	58	96	0	14	2	3	75	56	5
0	1	2	3	4	5	6	7	8	9

Le nombre qui, au sein d'un tableau, sert à repérer chaque valeur s'appelle **l'indice.** Chaque fois que l'on doit désigner un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément, entre croché.

La taille du tableau est le nombre de donnée contenu dans le tableau. Un tableau à une seule ligne est appelé vecteur

4.1.2. Déclaration

Un tableau est déclaré pour contenir des valeurs de même type (entier, réel, chaine, caractère…). Mélanger les variables est impossible. Pour le réaliser il nous faut créer d'autre structure. C'est ce que nous verrons bientôt. Mais pour l'instant nous parlons des tableaux à une dimension et statiques. C'est-à-dire que le nombre d'élément qu'il peut contenir est connu d'avance.

v crochet i est la donnée a l'indice i

La syntaxe est : NomDuTableau : **Tableau**[taille] **De Type des valeurs**

Exemple : Tab : **Tableau**[11] **De Entier** # Tableau de 11 entiers

Pour accéder à un élément quelconque dans un tableau, il suffit de connaitre son emplacement dans le tableau à travers l'indice de repérage.
Tab[5] pour accéder au 6ième élément du tableau nommé Tab. Donc **Tab**[5] vaut 2.

Les indices d'un tableau débutent par 0 dans ce cours comme c'est le cas de plusieurs langages de programmation à l'exemple de Python et C. Pour d'autre langage l'indice commence par 1.

4.1.3. Les Tableaux dynamique

Il arrive parfois que l'on ne connait pas à priori combien d'élément va contenir le tableau que l'on est en train de déclarer. L'une des solutions la plus facile et barbare serait de déclaré un tableau avec une très grande dimension (taille = 10 000). Humm ! Cette approche va susciter un gaspillage d'espace mémoire.

La solution que nous proposons dans ce cas de figure, c'est de déclarer le tableau et Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction de redimensionnement : Redim. On parle de tableau dynamique.

Tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable.

Syntaxe :

```
NomDuTableau : Tableau[] De Type des valeurs
Lire(n)
Redim NomDuTableau[n-1]
```

4.2. Les opérations sur les tableaux

Plusieurs opérations sont effectuées sur un tableau. Dans ce cours nous allons nous attarder sur les basiques. Les autres pourraient faire l'objet d'exercices, d'interrogations, et de devoirs sur table ou exposé.

4.2.1. Remplir un tableau

Le processus de remplissage d'un tableau consiste à stocker des valeurs dans un tableau. Nous allons remplir un tableau avec les 20 premiers termes de la suite de Fibonacci.

```
Procedure RemplirTabFibo(Tab :Tableau[19] De Entier)
Var
      i : Entier
Debut
      Tab[0] ← 0
      Tab[1] ← F1
      Pour i ← 2 A 19 Faire
         Tab[i] ← T[i-1]+T[i-2] # Stockage de F3 dans le tableau Tab
      FinPour
Fin
```

4.2.2. Affichage des éléments d'un tableau

Afficher les éléments d'un tableau consiste à parcourir ce tableau dans un sens unique et d'afficher chaque valeur de ce tableau. Nous allons utiliser la boucle Pour car le nombre d'opération est fini.

```
Procedure AfficheTab(Tab :Tableau[N] De Entier)
Var
      i : Entier
Debut
      Ecrire("Les éléments du tableau sont : ")
      Pour i ← 0 A N-1 Faire
           Ecrire(Tab|i]) # on affiche l'élément i
      FinPour
Fin
```

4.2.3. Maximum et minimum d'un tableau

Nous écrivons une procédure qui nous permettra d'afficher le maximum et le minimum contenu dans la table Tab. Deux approches sont possibles. La première est de parcourir le tableau tout en comparant un élément à tout le reste des éléments du tableau. Dès que l'on constat que l'élément comparé est plus grand que le précédent, automatiquement celui-ci devient le maximum. Nous ferons de même pour le minimum.

```
Procedure MaxMinTab(Tab :Tableau[N] De Entier)
Var
       Max, Min,i : Entier
Debut
       Max ← Tab[0]
       Min ← Tab[0]
       Pour i ← 0 A N-1 Faire
              Si(Tab[i]< Min)Alors
                     Min ← Tab[i]
              Si(Tab[0] > Max)Alors
                     Max ← Tab[i]
       FinPour
       Ecrire("Le maximum est : ",Max," Et le minimum est : ",Min )
Fin
```

La seconde approche consisterait à trier d'abord le tableau par ordre croissant ou décroissant et on récupère les extrémums en début et en fin du tableau.

4.2.4. Trier un tableau

Le tri à Bulle le tri bulle consiste a mettre la valeur la plus grande à l'extrémité puis reprendre l'opération jusqu'a ce qu'il n'y ai plus de changement

Un algorithme qui permet de tri un tableau. Comme son nom l'indique, ce tri fait remonter le plus faible élément en haut comme des bulles qui remonteraient à la surface du liquide.

Idée : parcoure le tableau et compare les couples d'élément successive, lorsque deux éléments successifs ne sont pas dans l'ordre ils sont échangés, après chaque parcoure du tableau, l'algorithme recommence l'opération.
Lorsque aucun échange n'a eu lieu pendant le parcoure, cela signifier que le tableau est bien trié et on arrêt la procédure.

```
Procedure TriBulle()
Var
    i, Temp : Entier
    Tab: Tableau[n]: De Entier
    Changement : Booleen

Debut
    Changement ← Vrai
    TantQue (Changement == Vrai) Faire
        Changement ← Faux
        Pour i ← 0 A n -1 Faire
            Si(Tab[i] > Tab[i + 1])Alors
                Temp ← Tab[i]
                Tab[i] ← Tab[i + 1]
                Tab[i + 1] ← Temp
                Changement ←  Vrai
            FinSi
         FinPour
    FinTantQue
Fin
```

Les autres tris

Plusieurs procédures existent pour trier un tableau. Chaque procédure à un avantage considérable en fonction du tableau dont l'on dont dispose. Ainsi avons-nous :

- Algorithme de tri stupide
- Algorithme de tri de Shell
- Algorithme de tri par tas
  (Pas au programme)
- Algorithme introsor
- Algorithme de tri à peigne
- Algorithme de tri rapide
- Algorithme de tri par insertion
- Algorithme de tri par sélection
- Algorithme de tri par fusion
- Algorithme de tri cocktail
- Algorithme de tri pair-impair

4.3. Travaux dirigés 5

Exercice 34

Ecrire un algorithme qui déclare et remplisse un tableau de sept (7) valeurs numériques en les mettant toutes à zéro.

Exercice 35

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

Exercice 36

Ecrivez un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs, qui devront être stockées dans un tableau. L'utilisateur doit donc commencer par entrer le nombre de valeurs qu'il compte saisir. Il effectuera ensuite cette saisie. Enfin, une fois la saisie terminée, le programme affichera le nombre de valeurs négatives et le nombre de valeurs positives.

Exercice 37

Une fonction qui permet de renvoyer la position d'un élément dans un tableau déjà saisie

Exercice 38

Ecrire une procédure qui permet de fusionner deux tableaux de tailles différentes et déjà saisies. Dans le tableau final, on ne souhaiterait pas voir un élément se répété.

Corrections

4.4. Les chaines de caractères

4.4.1. Bon à savoir :

La mémoire de l'ordinateur conserve toutes les données sous forme numérique. Il n'existe pas de méthode pour stocker directement les caractères. Chaque caractère possède donc son équivalent en code numérique : c'est le code ASCII (American Standard Code for Information Interchange - traduisez « Code Américain Standard pour l'Echange d'Informations »). Le code ASCII de base représentait les caractères sur 7 bits (c'est-à-dire 128 caractères possibles, de 0 à 127).

4.4.2. Définition

Une chaîne est une séquence de caractères ASCII. Les chaînes de caractères sont tous les mots ou plus généralement les assemblages de lettres et de chiffres.

4.4.3. Déclaration

Les chaînes de caractères sont manipulées de façon similaire aux tableaux (concaténation, parcours, indexation…).

```
Machaine : Chaine # Machaine peut supporter autant de caractères que je désire
Ou
Machaine[10]: Chaine # Machaine peut supporter que 10 caractères

Machaine ← "Bonjour"
```

Pour accéder à chaque élément de ma chaine il suffit de le manipuler comme un tableau :

```
Ecrire(Machaine[0]) # affiche "B"
```

```
Ecrire(Machaine[6]) # affiche "r"
```

4.4.4. Les opérations sur les chaines

Extraction

Extraire un caractère à une position i donnée

```
Machaine[i]
```

Extraction de plusieurs caractères. Ex : les caractères compris entre i et j

```
Machaine[i:j]
```

```
Machaine[:j]#Extrait du premier au j^ème caractère
```

```
Machaine[i:]#Extrait à partir du i^ème caractère jusqu'au derner
```

Concaténation

```
Machaine[i]+ " Tout le monde !" # "Bonjour Tout le monde !"
```

```
Machaine3 ← Machaine1 + Machaine2
```

La taille

```
len(Machaine)#Renvoie la taille de machaine qui est 6
```

Exercice 39

Un algorithme qui permet de calculer le nombre des voyelles, consonnes dans une phrase saisie par un utilisateur.

Exercice 40

Ecris une fonction compte le nombre de caractère contenu dans un texte qui lui sera passé en paramètre.

4.5. Les enregistrements

4.5.1. Définition

Contrairement aux tableaux qui sont des structures de données dont tous les éléments sont de *même type*, les enregistrements sont des structures de données dont **les éléments peuvent être de type différent** et qui se rapportent à la même **entité sémantique**. Les éléments qui composent un enregistrement sont appelés **champs**.

4.5.2. Déclaration d'un type structuré

Avant de déclarer une variable enregistrement, il faut avoir au préalable définit son type, c'est à dire **le nom** et **le type des champs** qui le composent. Le type d'un enregistrement est appelé type structuré. (Les enregistrements sont parfois appelés structures, en analogie avec le langage C et objet en programmation orientée objet).

Jusqu'à présent, nous n'avons utilisé que des types primitifs (caractères, entiers, réels, chaînes) et des tableaux de types primitifs. Mais il est possible de **créer nos propres types** puis de déclarer des variables ou des tableaux d'éléments de ce type.

Pour ce faire, il faut déclarer un nouveau type, fondé sur d'autres types existants. Après l'avoir défini, on peut dès lors utiliser ce type structuré tout autre type normal en déclarant une ou plusieurs variables de ce type. Les variables de type structuré sont appelées enregistrements.

La déclaration des types structurés se fait dans une section spéciale des algorithmes appelée Type, qui précède la section des variables (et succède à la section des constantes). Si l'algorithme comporte des sous-programmes, les types et les constantes sont déclarées en dehors du programme.

Syntaxe

```
Type
      Structure NomType
            NomChamp1: TypeChamp1
                  …
            NomChampN: TypeChampN
      FinStruct
```

➔ Représentation

NomType			
Champs	NomChamp1	…	NomChampN

➔ Exemple :

```
Type
      Structure Personne
            Nom : Chaine
            Prenoms : Chaine
            Age : Entier
      FinStruct
```

4.5.3. Déclaration d'un enregistrement à partir d'un type structuré

Une fois qu'on a défini un type structuré, on peut déclarer des variables d'enregistrements exactement de la même façon que l'on déclare des variables d'un type primitif. La syntaxe est la même.

➔ Syntaxe :

```
Var
    NomVariable : TypeEnregistrer
```

Exemple

```
Var
        Pers1, Pers2, Pers3 : Personne
```

➔ Représentation

Les enregistrements sont composés de plusieurs zones de données, correspondant aux champs

	Pers1.N	Pers1.Pren	Pers1.Ag
Pers1	Loua	Emaüs Dany	12

4.5.4. Manipulation d'un enregistrement

La manipulation d'un enregistrement se fait au travers de ses champs. Comme pour les tableaux, il n'est pas possible de manipuler un enregistrement globalement, sauf pour affecter un enregistrement à un autre de même type (ou le passer en paramètre). Par exemple, pour afficher un enregistrement il faut afficher tous ses champs un par un.

Accès au champ d'enregistrement

Alors que les éléments d'un tableau sont accessibles par l'intermédiaire de leur indice, les champs d'un enregistrement sont accessibles à travers leur nom, grâce à l'opérateur point « . »

```
NomEnregistrement.NomChamp

Représente la valeur mémorisée dans le champ de l'enregistrement
```

Par exemple pour accéder à la valeur de l'âge du personnage Pers1 on écrira :

```
SonAge ← Pers1.age  # SonAge vaut 12
```

Remarque : la lecture d'une telle expression se fait de la droite à la gauche : l'âge de la personne 1.

Attention : le nom d'un champ est TOUJOURS précédé du nom de l'enregistrement auquel il appartient. On ne peut pas trouver un nom de champ tout seul, sans indication de l'enregistrement.

Exercice

Exercice 41

On désire enregistrer N étudiants par leurs noms, prénoms, leur sexe et leur âge. Ecrire un algorithme qui effectue cet enregistrement.

Correction 41

```
Algorithme EnregistrementEtudiant
#Déclaration de la structure
Type
Structure Etudiant
Nom : Chaine
Prenoms : Chaine
Sexe : Caractere #(M ou F)
Age : Entier
FinStruct
Var
       Tab : Tableau[] : De Etudiant
       Etud : Etudiant
       i,N : Entier
Debut
       Ecrire("Entrez le nombre d'étudiant SVP : ")
       Lire(N)
       Redim Tab[N]
       Pour 0 ← 1 A N-1 Faire
              Ecrire("Entrez l'étudiant N°",i+1, " Svp : ")
              Ecrire("Nom :  ")
              Lire(Etud.Nom) #Enregistrement du champ Nom
              Ecrire("Prénoms :  ")
              Lire(Etud.Prenoms) #Enregistrement du champ Prenoms
              Ecrire("Age :  ")
              Lire(Etud.Age) #Enregistrement du champ Age
              Ecrire("Sexe(M ou F) : ")
              Lire(Etud.Sexe) #Enregistrement du champ Sexe
              Tab[i] ← Etud
       FinPour
FinProcedure
```

4.5.5. Un enregistrement comme champ d'une structure

Supposons que dans le type Etudiant, nous ne voulions plus l'âge de la personne, mais sa date de naissance. Une date est composée de trois variables (jour, mois, année) indissociables2. Une date correspond donc à une entité du monde réel qu'on doit représenter par un type enregistrement à 3 champs.
Si on déclare le type date au préalable, on peut l'utiliser dans la déclaration du type personne pour le type de la date de naissance.

Un type structuré peut être utilisé comme type pour des champs d'un autre type

2 en supposant que le type Date n'existe pas

```
TYPE
      Structure LaDate
            Jour: Entier
            Mois: Chaîne
            Annee: Entier
      FinStuct

      Structure Personne
            Nom : chaîne
            DtNaissance : LaDate
      FinStruct
```

Pour accéder à l'année de naissance d'une personne, il faut utiliser deux fois l'opérateur « . »

```
Var
      Pers1 : Personne

Annee_de_Naissance←Pers1.DtNaissance.Annee
```

Il faut lire une telle variable de la droite à la gauche : l'année de la date de naissance de la personne 1.

4.5.6. Les tableaux d'enregistrements (ou tables)

Il arrive souvent que l'on veuille traiter avec non pas un seul enregistrement mais plusieurs (cas de l'exercice 40). Par exemple, on veut pouvoir traiter un groupe d'étudiant. On ne va donc pas créer autant de variables du type Etudiant qu'il y a d'étudiants. On va créer un tableau regroupant toutes les étudiants du groupe. Il s'agit alors **d'un tableau d'enregistrements**, autrement appelé « **table** ».

```
Type
      Structure Etudiant
            Nom : Chaine
            Prenoms : Chaine
            Sexe : Caractere #(M ou F)
            Age : Entier
      FinStruct
Var
       Tab : Tableau[N] : De Etudiant
```

⚠ Représentation

Tab[0] = Etudiant1				Tab[1]=Etudiant2				...
Nom1	Prenoms1	Sexe1	Age1	Nom2	Prenoms2	Sexe2	Age2	...

Ou plus mieux encore : **une table**

	Nom	Prenoms	Sexe	Age
Tab[0]	Koffi	Rachelle Amoin	F	24
Tab[1]	Bamba	Souhalio Kobra	M	18
...	...	...	...	...
Tab[N-1]	Kablan	Raoule David	M	18

⚠ Attention !

~~Tab.Nom[3]~~ n'est pas valide.
Pour accéder au nom de la quatrième étudiant du tableau, il faut écrire Tab[3].Nom

4.6. Travaux dirigés 6

Exercices

Exercice 42

Déclarer des types qui permettent de stocker :

1. Un joueur de basket caractérisé par son nom, sa date de naissance, sa nationalité, et son sexe
2. Une association de joueurs de basket

Exercice 43

Un algorithme qui permet de comparer deux temps et afficher le plus grand.

Exercice 44

a. Construire un type **Complexe** qui représente le type des nombres complexes.
b. Ecrire une procédure ou fonction **Conj()** qui prend en paramètre un nombre complexe puis renvoie son **conjugué.**

c. Ecrire une procédure ou fonction **Module()** qui prend en paramètre un nombre complexe puis renvoie son **module**.

d. Ecrire une procédure ou fonction **Som2()** qui prend en paramètre deux nombres complexes et qui permet de renvoie leur **somme**.

e. Ecrire une procédure ou fonction **Prod2()** qui prend en paramètre deux nombres complexes et qui permet de renvoie leur **Produit**.

correction