Les instructions de base

Un premier algorithme

Algorithm: Square of a number

```
#Déclaration des entrées et des sorties, réservation de la mémoire
Input a number : real
Output The square of the number : real
begin
    print("Enter a real number :")*   #Préparation du traitement
    read(value)                       #Instruction pour lire ...
    square ← value × value            #Traitement, opération ...
    print("the square of ", value)
    print("is ", square)              #Affichage du résultat ...
end
```

Commenter : anglais, déclarations, entrées, sorties, traitements

Résumé

- Titre + commentaire de l'algorithme
- Déclaration des variables d'entrée ou de sortie avec leur type
- Début
- écriture de messages, lecture des entrées
- traitement
- affichage des sorties
- Fin

Les trois étapes d'un algorithme

1. **Préparation du traitement**
   - données nécessaires à la résolution du problèmes
   - paramètres
2. **Traitement**
   - résolution pas à pas, après décomposition en sous-problèmes si nécessaire
   - opérations mathématiques ou sur les chaînes de caractères, ou sur les fichiers, ou des données
3. **Edition des résultats**
   - sortie standard : l'écran
   - sortie dans un fichier

Schéma à mémoriser et à utiliser à chaque fois qu'on doit programmer.

Déclarer une variable

variable ⟨ liste d'identificateur ⟩ : **type**

- **Fonction**
  - réserver l'espace mémoire pour stocker les données
  - définir le type : entiers, réels, caractères, booléen
- **Exemples**

variables	value, square	: real
	i, index	: integer
	flag	: boolean
	name, city	: strings (chaîne de caractères)

Séparateur de variables : la virgule (,), déclaration : deux points (:)

- On préfèrera plutôt déclarer le type de la variable : **Input** ou **Output**
- La mise en forme et la convention peuventt légèrement varier suivant les auteurs,

Afficher ou imprimer des données, messages, résultats

print (⟨ liste de variables, chaines de caractères, ... ⟩)

- **Fonction**
  - imprimer ou afficher une ou plusieurs informations ou messages
  - mettre des messages clairs et instructifs
- **Exemples**

  print ("la valeur de l'entrée est ", value)
  print ("j'habite à ", city, "qui se trouve en ", country)

- Notez les "" à cause de l'apostrophe utilisée en français
- Affichage de chaînes de caractères et de contenu de variables.

Saisir une donnée

read (⟨ liste de noms de variables ⟩)

- **Fonction**
  - placer en mémoire les données entrées
  - il faut toujours mettre un message avant pour préciser les entrées à l'utilisateur
- **Exemples**

  print ("Entrer un nombre réel positif : ")
  read (value)
  read (i, index)
  read (city, country)

Définir des constantes et leur valeur

parameter (⟨ identificateur⟩ : type) ← ⟨ valeur ou expression⟩

- **Fonction**
  - un paramètre ou une constante a sa valeur fixée (figée)
  - son traitement peut être légèrement différent d'une variable
- **Exemples**
  parameter (PI : real) ← 3.1415927
  parameter (IMAX : integer) ← $2 \times 11 + 5$

- une constante est toujours un paramètre d'entrée
- par conventionn, le nom de la constante est **en lettres capitales**

Affecter une valeur à une variable

⟨ identificateur⟩ ← ⟨ expression ou valeur ou identificateur ⟩

- **Fonction**
  - attribution d'une valeur à une variable dans la mémoire de l'ordinateur
  - attention au sens de la flèche.
- **Exemples**

  value ← 3.1415927
  result ← 2 × value + 5
  value ← 2 × value + 5
  name ← "no_one"

- ← est l'opérateur d'affectation en algorithmique
- il peut changer d'un langage de programmation à un autre
- c'est extrêmement souvent "="

Traitements conditionnels

Effectuer une opération sous condition "si"

Algorithm: Condition "si"

Une condition est un test dont le résultat est un booléen "vrai" ou "faux" (true/false)

if cond1 **then**
| `print`(*"do something"*)
else
| `print`(*"do something else"*)
end

if cond1 **then**
| cond1 is true
else if cond2 **then**
| cond1 is false but cond2 is true
else
| all is wrong
end

conditions imbriquées

Algorithm: Condition "si" emboitée

On peut ajouter des niveaux dans les tests

```
begin
    if cond1 then                                   #first level
        action 1
    else if cond2 then
        if cond 3 and cond 4 then                   #second level
            action 2
        else
            action 3
        end
    else
        action 4
    end
end
```

Ne pas faire plus de 2 niveaux, sinon utiliser des fonctions.

Sélection par choix multiple "selon"

Algorithm: Condition "selon"

```
switch value do
    case (1, 2, 3) do                          #list of numbers
        print(" lower equal to 3")
    case 4 do
        print("is is four")
    otherwise do                               #default case
        print("is is another number !")
end
```

- On peut le faire avec des mots (chaîne de caractères)
- Très utile pour faire des menus
- Cette instruction n'existe pas dans tous les langages (bientôt dans Python)

Boucle "Pour"

Algorithm: Boucle "Pour"

```
for counter ← init_value to final_value by step do    #counter is an integer
    do something
end
for value ∈ list do                                   #value in a given list
    do something else
end
```

- Répétition d'une tâche un certain nombre de fois.
- Ne fonctionne que **si on connaît à l'avance** combien d'itérations seront effectuées
- Ne jamais modifier la valeur du compteur → ERREUR
- La variable de la boucle peut être comprise dans une liste de valeurs ou d'éléments (très puissant dans Python)

Boucle "Tant que"

Algorithm: Boucle "tant que"

```
i ← 2                                  #Initialisation required
while i ≤ 1000 do                      #any condition .........
  | i ← i × 5
end
```

- Répétition d'une tâche un certain nombre de fois tant qu'une certaine condition est vérifiée
- Très utilisé **si on ne connaît pas à l'avance** combien d'itérations seront effectuées
- On doit modifier la condition dans la boucle si on veut en sortir
- La condition est testée **au début** de la boucle, penser à l'initialiser
- La boucle peut ne jamais être effectuée
- À utiliser avec soin et en connaissance

Boucle "Répéter"

Algorithm: Boucle "Répéter"

```
i ← 2                                   #Initialisation required
repeat
 |  i ← i × 5
until  i ≤ 1000
```

- Répétition d'une tâche un certain nombre de fois tant qu'une certaine condition est vérifiée
- Très utilisé si on ne connaît pas à l'avance combien d'itérations seront effectuées
- On doit modifier la condition dans la boucle si on veut en sortir
- La condition est testée **à la fin** de la boucle, penser à l'initialiser
- La boucle est effectuée au moins une fois
- À utiliser avec soin et en connaissance

Choix des boucles

En fonction

- du nombre d'itérations connu ou pas à l'avance
- des tests simples ou complexes
- des valeurs dans une liste
- de l'application : créer un menu
- du besoin d'associer avec plusieurs boucles
- du besoin d'ajouter des tests "si"
- de l'importance des conditions qui peuvent être complexes avec des "et","ou", "non", $<$, $>$, $=$, $\leqslant$, $\geqslant$, $\neq$
- rechercher l'efficacité et la simplicité

Itérateurs très puissants en Python

Logique propositionnelle

Quelle logique ?

- La logique est une façon de formaliser un raisonnement pour résoudre un problème
- Il existe différentes logiques.
- **La logique propositionnelle** est un modèle mathématique qui permet de raisonner en fonction de la nature vraie ou fausse d'expressions logiques

Exemple de conditions logiques

- **while** value $\leqslant$ THRESHOLD **do** ...
- **while** (value $\neq$ LIMIT) **and** (iteration $\leqslant$ ITER_MAX) **do** ...
- **if** (value $\leqslant$ LIMIT) **or** (value % 21 $=$ 0) **do** ...
- $2 + 2 = 6$ $\quad$ $(2 + 2 == 6)$ $\quad$ # test, pas affectation !
- $2 \times 5 = 10$ $\quad$ $(2 \times 5 == 10)$
- animal $=$ "cat"
- $x > y$

Remarques

- Pour tester l'égalité logique de deux choses on utilise souvent le symbole "==" pour différentier de l'égalité mathématique ("=").
- La réponse à ces conditions sera toujours **true** ou **false**
- Dans la plupart des langages on assigne 1 pour **true** et 0 pour **false**

Éléments de logique propositionnelle

Regarder la page web :
`https://fr.wikipedia.org/wiki/Alg%C3%A8bre_de_Boole_(logique)`
rechercher dans google "algèbre de Boole" et regarder la page wikipédia.
On a trois éléments :

Formule

Les expressions logiques sont composées de variables propositionnelles et de connecteurs logiques.

Variable propositionnelle

une proposition considérée comme indécomposable (*minimaliste*)

Connecteurs logiques

`https://fr.wikipedia.org/wiki/Liste_de_symboles_logiques`

Connecteurs logiques

sens	Fr	En	logique info	Langage	Boole
négation	non	not	$\neg$	$\sim$, !, not	$\overline{condition}$
conjonction	et	and	$\wedge$	&&	$\cdot$
disjonction	ou	or	$\vee$	||	$+$
équivalent			$\Leftrightarrow$	==	$=$
implicitation			$\Rightarrow$	*	

Calcul d'une formule : table de vérité

s	=	(a or b) and not (a and c)	langage
	=	$(a \vee b) \wedge \neg(a \wedge c)$	informatique
	=	$(a + b) \cdot \overline{a \cdot c}$	Boole

a	b	c	condition
		false, 0	$a + b$
1			$\bar{c}$
0			b
	0		$a \cdot \bar{c}$

Table de vérité

Toutes les combinaisons de a, b et c sont testées pour les valeurs vraies ou faux (ou 0 ou 1).

Génération de la table de vérité

$s = (a + b) \cdot \overline{(a \cdot c)}$

- 3 variables : $2^3 = 8$ cas
- Apprendre les règles de base

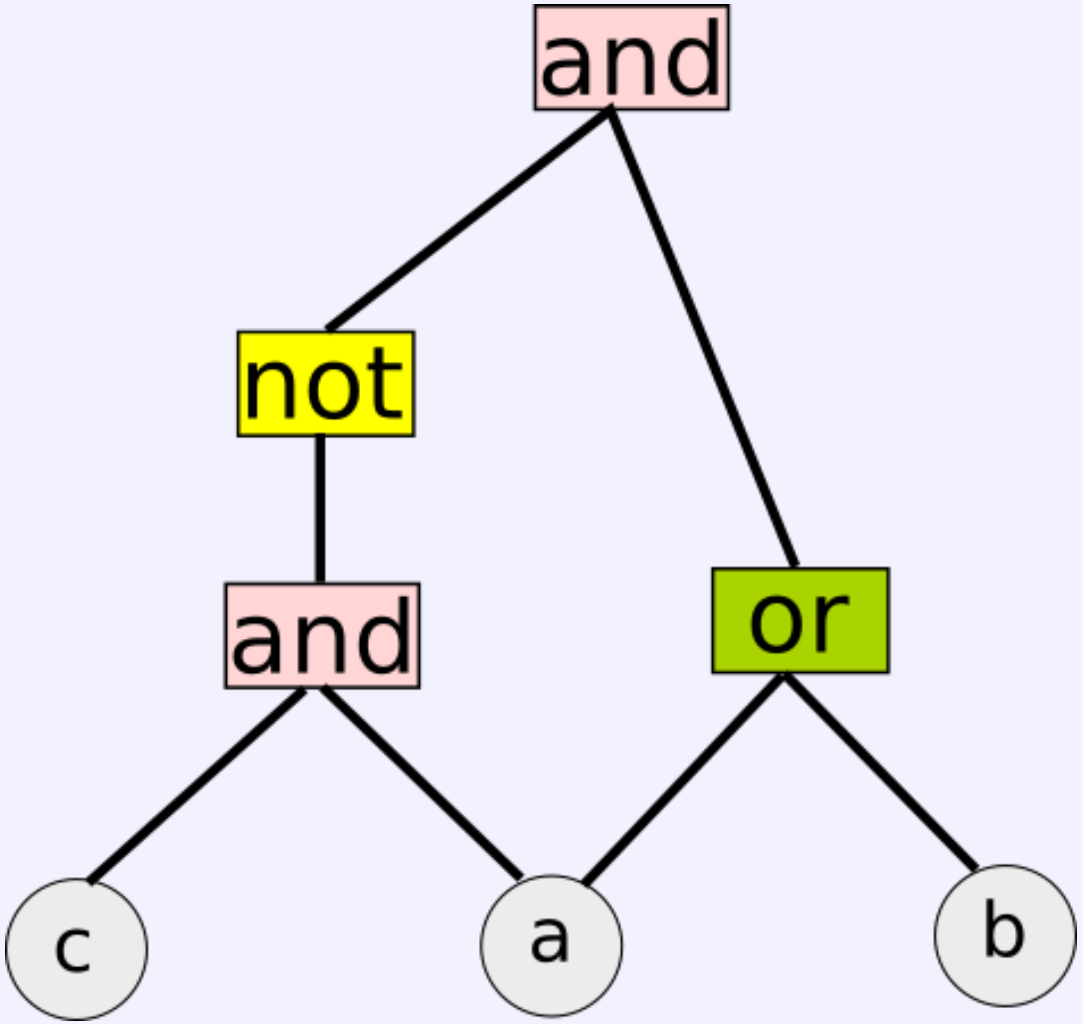
$0 + 0 = 0$
$0 \cdot 0 = 0$

$1 + 0 = 0 + 1 = 1$
$1 \cdot 0 = 0 \cdot 1 = 0$

$1+1 = 1$
$1 \cdot 1 = 1$

n	a	b	c	a+b	a · c	s
0	0	0	0			
1	0	0	1			
2	0	1	0			
3	0	1	1			
4	1	0	0			
5	1	0	1			
6	1	1	0			
7	1	1	1			

Représentation d'une formule : arbre syntaxique



Ne pas en discuter

Faire l'analyse syntaxique de l'arbre :

- pour la compréhension
- pour la simplification

algèbre de Boole

Commutativité

- $a + b = b + a$
- $a \cdot b = b \cdot a$

Associativité

- $(a + b) + c = a + (b + c) = a + b + c$
- $(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$

Distributivité

- $(a + b) \cdot c = a \cdot c + b \cdot c$ très utile
- $(a \cdot b) + c = (a + c) \cdot (b + c)$ peu utile

Résultats évidents

- $a + \bar{a} = 1$ toujours vrai (**tautologie**)
- $a \cdot \bar{a} = 0$ toujours faux (**contradiction**)
- $a + a = a$, $a \cdot a = a$ toujours vrai (**tautologie**)

Loi de Morgan

Fondamentalement important et utile

Négation

- $\overline{a+b} = \bar{b} \cdot \bar{a}$
- $\overline{a \cdot b} = \bar{b} + \bar{a}$

Formule test

$$s = (a+b)\cdot\overline{a \cdot c} = (a+b)\cdot(\bar{a}+\bar{c}) = a\cdot\bar{a}+a\cdot\bar{c}+b\cdot\bar{a}+b\cdot\bar{c} = a\cdot\bar{c}+b\cdot\bar{a}+b\cdot\bar{c}$$

donc

$$\begin{array}{lcl} a = 1 & \Rightarrow & s = \bar{c} + b \cdot \bar{c} = \bar{c} \\ a = 0 & \Rightarrow & s = b + b \cdot \bar{c} = b \\ b = 0 & \Rightarrow & s = a \cdot \bar{c} \\ b = 1 & \Rightarrow & s = \bar{a} + \bar{c} \\ c = 0 & \Rightarrow & s = a + b \\ c = 1 & \Rightarrow & s = b \cdot \bar{a} \end{array}$$

Formule équivalente

Fondamentalement important et utile

Implication / Equivalence

- $a \Rightarrow b = \bar{a} + b$
- $a \Rightarrow b = \bar{b} \Rightarrow \bar{a}$

Faire la table de vérité pour vérifier et comprendre ces équivalences.

Application à l'algorithmique

Définir la condition de fin d'une boucle : **while** ...

- On sort de la boucle si $cond_1 + cond_2$
- Donc on reste dans la boucle si $\overline{cond_1 + cond_2} = \overline{cond_1} \cdot \overline{cond_2}$
- Attention la négation de "supérieur ou égal" est "strictement inférieur" : $\overline{a \leqslant b} \quad = \quad a > b$

Usage très fréquent en programmation

- pour simplifier des conditions complexes
- pour vérifier la validité d'une condition : $(x < 10) \cdot (x > 100)$ est toujours faux par exemple,
- pour écrire la négation d'une condition

Variable booléenne (boolean type)

Toute variable donnant le résultat d'une proposition, d'un test. Elle a toujours soit la valeur (true,1) ou (false, 0).

Exemple d'algorithme → maison

Distribution de bonbons

1. Le soir d'Halloween, un groupe de N_e enfants a récupéré N_b bonbons qu'ils souhaitent se répartir le plus équitablement.
2. Ils se mettent autour d'une table et commencent à prendre chacun N_i bonbons.
3. Lorsqu'il ne reste pas assez de bonbons, ils essaient d'en prendre $N_i - 1$.
4. Puis chaque fois qu'il n'y a plus assez de bonbons, ils essaient d'en prendre encore 1 de moins et ainsi de suite.
5. Au bout d'un moment, il reste moins de bonbons que d'enfants.

Faire un algorithme pour trouver :

1. Combien de bonbons obtient chaque enfant ?
2. Combien de tours ont-il effectué ?
3. Combien reste-t-il de bonbons ?

L'algorithme est rapide si on réfléchit bien.

Solution

- décrire les entrées / sorties
- mettre des commentaires intelligents et les aligner
- mettre des espaces entre les opérateurs

Algorithm: Candy distribution

```
Input ne : integer                          #Children number ..........
Input nb : integer                          #Candy number .............
Input ni : integer                          #initial candy distribution
Output round : integer                      #round number .............
Output m : integer                          #candy number by child ....
Output r : integer                          #remaining candies ........
begin
    print("Enter ne, nb, ni :")*        #on peut allonger le message ...
    read(ne, nb, ni)
    r ← nb % ne                         #Le reste de la division entière
    m ← nb / ne                         #/ : division entière ..........
    round ← m / ni + 1                  #il faut un tour supplémentaire
    print("Results : candies/child, rounds, remaining", m, round, r)
end
```

Distribution de tous les bonbons

Complication

L'algorithme précédent n'est pas réaliste : les enfants voudront distribuer tous les bonbons, et chacun veut distribuer les bonbons. On propose donc une modification.

- À chaque tour l'enfant qui distribue tourne.

  Celui qui distribue se sert toujours en dernier.

  On distribue N_i bonbons au premier tour, et à chaque tour ce nombre diminue de 1.

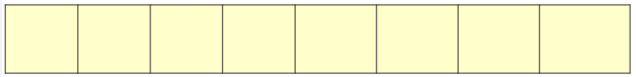
  Lorsque $N_i = 1$ on ne distribue plus qu'un seul bonbon par tour.

  Avec ce processus, il est maintenant possible de distribuer tous les bonbons, mais tous les enfants n'auront pas le m*me nombre de bonbons.
- Déterminer le nombre de tours, le nombre de bonbons par enfants et qui a distribué en dernier.
- Les enfants seront nommés par un entier de 1 à N_e, une fois pour toute.
- Cette version est plus complexe et nécessite une bonne dose de réflexion et probablement l'emploi de boucles et de listes ou tableaux.

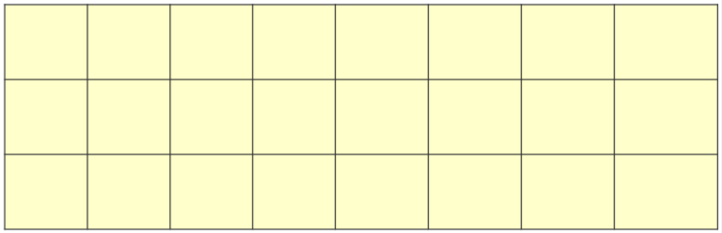
Tableaux - listes

Tableau : structure pour des données de même type

- Besoin de conserver des données de même types pour leur faire subir un traitement identique
- **tableau** à une dimension



- **tableau** à deux dimensions



- Contenu du tableau : des nombres entiers ou réels, des caractères ou chaînes de caractères, des valeurs booléennes
- Exemple : un agenda est une liste de nom, prénom, ville, téléphone, adresse, ...
- Une **liste** sera un tableau à une dimension, mais dans lequel les données ne seront pas nécessairement de même type.

Tableau : traitement des données

Opérations sur un tableau

- déclarer et créer un tableau
- écrire des valeurs
- lire des valeurs
- rechercher l'existence d'une valeur donnée
- modifier une ou plusieurs valeurs
- trier les valeurs numériques
- ajouter ou retirer des éléments
- effectuer des opérations entre tableaux : addition, multiplication, ..., comparaisons

Tableau : définition

Indexation des éléments

- un indice (nombre entier positif) permet d'accéder à un élément du tableau
- le premier élément est l'indice 1 (mais dans certains langages cela commence à 0)
- la place prise par le tableau est fonction du nombre d'octets pris par chaque élément en mémoire
- la longueur (taille) du tableau est le nombre d'éléments, à ne pas confondre avec sa forme ($n \times m$ en 2D)

1	2	3	4	5	6	Matlab
-5	4	2.3	1	-1.4	101	
1	0	1	1	0	0	
r	e	q	u	i	n	
0	1	2	3	4	5	Python

Notations (en fonction des conventions) :

- 1 D : $x(1)$ ou $x[1]$
- 2 D : $z(2,3)$ ou $z[2,3]$

Dans les langages de programmation $x(1)$ est différent de $x[1]$.

Déclarer une tableau

variable ⟨ liste d'identificateur ⟩ : **array**[ ] type

1. on peut choisir le début et la fin de l'indexation [0, 9] ou [1, 10] ou [10, 20] ou [-5, 5]
2. Ne pas confondre l'indice du tableau i avec la valeur $x(i)$.
3. L'indice d'un tableau est toujours un nombre entier

Exemples

- **variable** tab : **array** [1, 10] integer
- **variable** x : **array** [10] real
- **variable** y : **array** [0 :3, 9] real
- **variable** z : **array** [0 :3, 100 :200]

Dans certains langages, il n'est pas toujours nécessaire de déclarer le tableau avant l'**usage**. Voir les spécificités pour le Python ...

Exemple d'algorithme

```
Algorithm: enter array values
Output n : integer
Output tab : array [1, 10] integer
variable : i : integer
begin
    print("Enter number of elements n ⩽ 10 :")
    read(n)
    if n > 10 then
        print("error : n > 10")
    else
        for i = 1 to n do
            print("enter element (integer) ", i)
            read(tab[i])
            tab[i] ← 3 * tab[i] + 2
        end
    end
    print("Output array : ", tab)
end
```

Sous-algorithmes, modularité

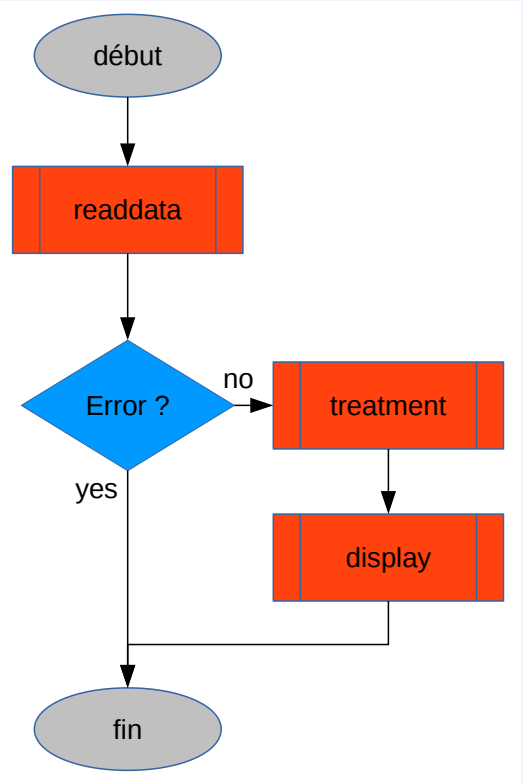
Résumé, exemple

- simplifier l'écriture pour être plus compréhensible
- déporter des parties de l'algorithme ailleurs
- l'algorithme est plus lisible, donc plus clair et facile à modifier ou implémenter

Algorithm: enter array values

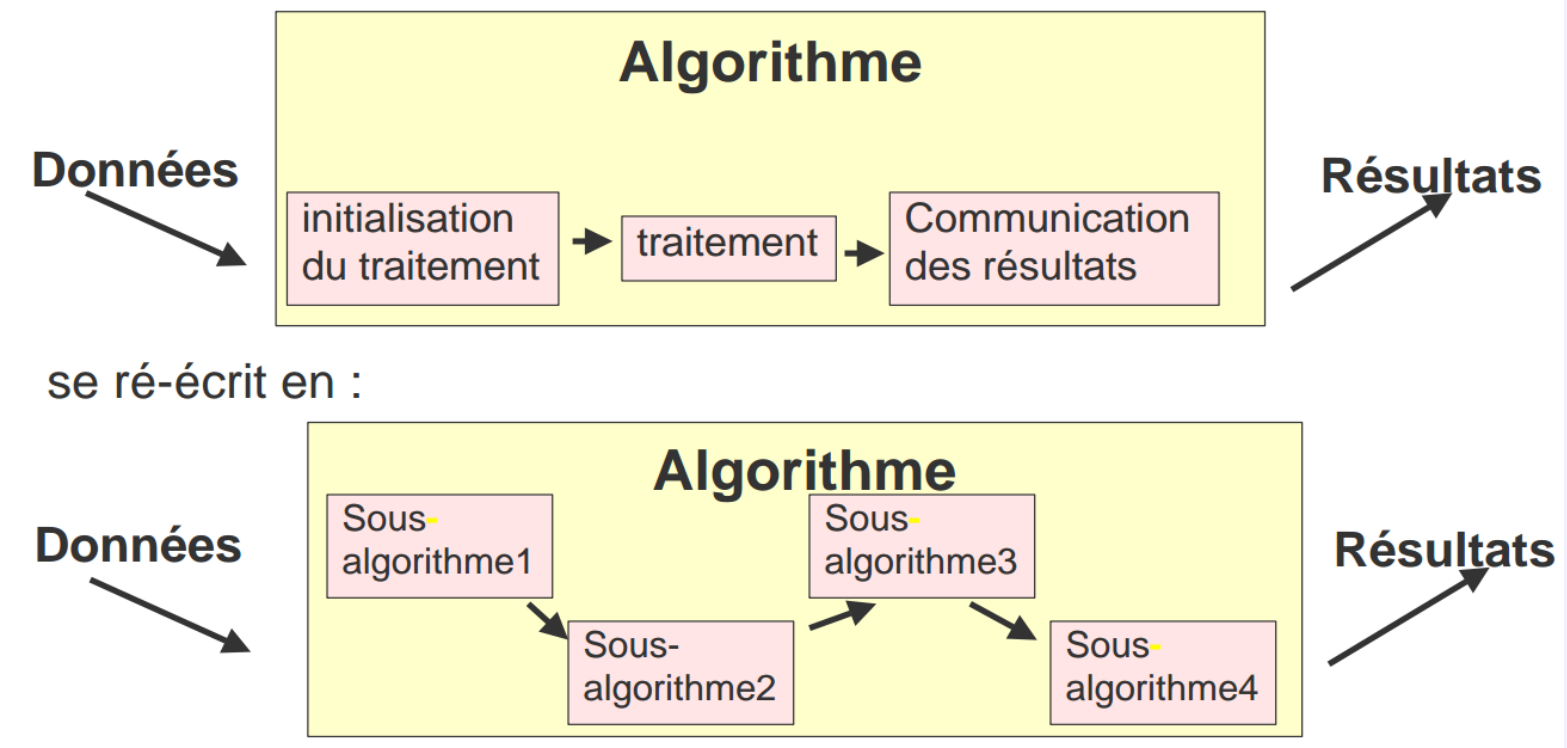
Output *n* : integer
Output *tab* : array [1, 10] integer

```
0 begin
1     readdata(n, error)
2     if not error then
3         treatment(n, tab)
4         display(n, tab)
5     end
6 end
```



Les nouvelles fonctions sont alors appelées des **sous-algorithmes**

Réorganisation algorithmique



- le sous-algorithme effectue le traitement sur des éléments bien définis et délimités, et si possible indépendamment du contexte de l'algorithme appelant
- un sous-algorithme peut en appeler un autre.

Arguments, paramètres

Algorithm: enter array values

Output *n* : integer
Output *tab* : array [1, 10] integer

```
0 begin
1     readdata(n, error)
2     if not error then
3         treatment(n, tab)
4         display(n, tab)
5     end
6 end
```

Algorithm: readdata (m, erreur)

{read the size of the input array}
Output *m* : integer
Output *erreur* : boolean

```
0 begin
1     erreur ← false
2     print("Enter number of
        elements n ⩽ 10 :")
3     read(m)
4     if m > 10 then
5         print("error : n > 10")
6         erreur ← true
7 end
```

- *n*, *error* sont les arguments lorsque le sous-algorithme `readdata` est appelé
- *m*, *erreur* sont les paramètres du sous-algorithme `readdata`
- les variables sont **muettes** dans les sous-algorithmes

Sous-algorithmes identiques

Algorithm: readdata (m, erreur)

{read the size of the input array}
Output *m* : integer
Output *erreur* : boolean

```
0 begin
1     erreur ← false
2     print("Enter number of elements n ⩽ 10 :")
3     read(m)
4     if m > 10 then
5         print("error : n > 10")*
6         erreur ← true
7 end
```

Algorithm: readdata (n, error)

{read the size of the input array}
Output *n* : integer
Output *error* : boolean

```
0 begin
1     error ← false
2     print("Enter number of elements n ⩽ 10 :")
3     read(n)
4     if n > 10 then
5         print("error : n > 10")
6         error ← true
7 end
```

- les variables sont **muettes** dans les sous-algorithmes
- **initialiser les paramètres de sortie** pour toujours avoir une valeur
- Il doit y avoir **autant d'argument en appel** qu'il y a de paramètres dans le sous-algorithme

Sous-algorithmes : règle de bases

1. les variables sont muettes dans les sous-algorithmes : les arguments et les paramètres ne sont pas sensés porter les mêmes noms.
2. initialiser les paramètres de sortie pour toujours avoir une valeur à la sortie du sous-algorithme
3. Il doit y avoir autant d'argument en appel qu'il y a de paramètres dans le sous-algorithme
4. Les paramètres et les arguments doivent être écrits dans le bon ordre
5. Il faut faire correspondre les types des arguments et des paramètres
   à un paramètre réel doit correspondre un argument réel
6. Il faut respecter les dimensions lorsqu'un paramètre est un tableau
7. Un paramètre d'entrée doit avoir une valeur lors de l'appel du sous-algorithme, même si cette valeur peut être modifiée ultérieurement
8. Bien formaliser les sous-algorithme en déclarant les entrées, les sorties, les variables locales (internes)
9. Toujours indiquer par un commentaire le rôle du sous-algorithme

Sous-algorithme : fonction

Une fonction (function) est un sous-algorithme qui retourne une valeur :

```
Function treatment(n)
    { modifying the array}
    Input n : integer                          #Maximal length of the array
    Output tab : array [1, 10] integer
    variable i : integer                       #Local variable for the loop
    begin
        for i = 1 to n do
            print("enter element (integer) ", i)
            read(tab[i])
            tab[i] ← 3 * tab[i] + 2
        end
        return tab
    end
```

Sous-algorithme : procédure

Une procédure (procedure) est un sous-algorithme qui dans son appel contient à la fois les paramètres d'entrées et de sorties

```
Procedure readdata(nmax, n, error)
    { read the size of the input array}
    Input nmax : integer                    #Maximal length of the array
    Output n : integer          #length of the filled part of the array
    Output error : boolean
    begin
        error ← false
        print("Enter number of elements n ⩽", nmax, " :")
        read(n)
        if n > nmax then
            print("error : n > ", nmax)
            error ← true
        return n, error
    end
```

- En Python, la procédure n'existe pas, c'est une fonction qui peut retourner plusieurs valeurs.
- Par contre elle existe en Fortran.