Chapitre III

Bases d'Algorithme & Programmation en C

Sommaire

Chapitre II : Bases de l'Algorithme & Programmation en C....4
II.1. Concept d'un algorithme....4
II.2. La démarche d'analyse d'un problème....5
II.3. Données, Identificateurs, Types de données simples....5
II.3.1. Notion d'identificateur....6
II.3.2. Constantes et variables....6
II.3.3. Types de données....7
II.4. Structure d'un algorithme / programme....7
II.4.1. Déclarations....8
II.4.2. Corps....10
II.5.Types d'instructions....10
II.5.1. Instructions d'Entrées/Sorties (Lecture / Écriture)....10
II.5.1.1. Entrées (Lecture)....10
II.5.1.1. Sorties (Écriture)....11
II.5.2. Instruction d'affectation....12
II.5.3. Structures de contrôles....12
II.5.3.1. Structures de contrôle conditionnelle....13
a. Test alternatif simple....13
b. Test alternatif double....13
II.5.3.2. Structures de contrôle répétitives....14
a. Boucle Pour (For)....14

---

c. Boucle Tant-que (while).....15
c. Boucle Répéter (Repeat).....15
II.6. Représentation en organigramme.....16
II.6.1. Les symboles d'organigramme.....16
II.7.2. Représentation des primitives algorithmiques.....17
II.7.2.1. L'enchaînement.....17
II.7.2.2. La structure alternative simple.....17
II.7.2.3. La structure alternative double.....18
II.7.2.4. La structure itérative POUR (Boucle POUR).....18
II.7.2.5. La structure itérative Tant-Que (Boucle Tant-Que).....19
II.7.2.6. La structure itérative Répéter (Boucle Répéter).....20
II.7. Exemples d'Application.....21
II.7.1. Exemple 1 : Permutation de valeurs de deux variables.....21
II.7.2. Exemple 2 : Somme de deux variables.....23
II.7.3. Exemple 3 : Nombre pair ou impair.....24
II.7.4. Exemple 4 : Afficher tous les nombres divisibles par *n*.....26
II.7.5. Exemple 5 : *Rechercher les diviseurs d'un nombre n*.....29

Cours Elearning :
https://elearning.univ-bejaia.dz/course/view.php?id=14316

Page facebook :
https://www.facebook.com/InitiationAlgoProgrammation/

La chaîne Youtube :
https://www.youtube.com/c/AlgoProgrammation1èreAnnéeTechnologie

La playlist sur le langage C :
https://youtube.com/playlist?list=PLwHHAvorm5F-tL9EXDEH0miOKmAj7iUTU

Adapté par : Redouane OUZEGGANE

rouzeggane@gmail.com - redouane.ouzeggane@univ-bejaia.dz

Chapitre II : Bases de l'Algorithme & Programmation en C

II.1. Concept d'un algorithme

– Le mot « Algorithme » est inventé par le mathématicien « AL-KHÂWARIZMÎ ». Un Algorithme est l'énoncé d'une séquence d'actions primitives réalisant un traitement. Il décrit le plan ou les séquences d'actions de résolution d'un problème donné.

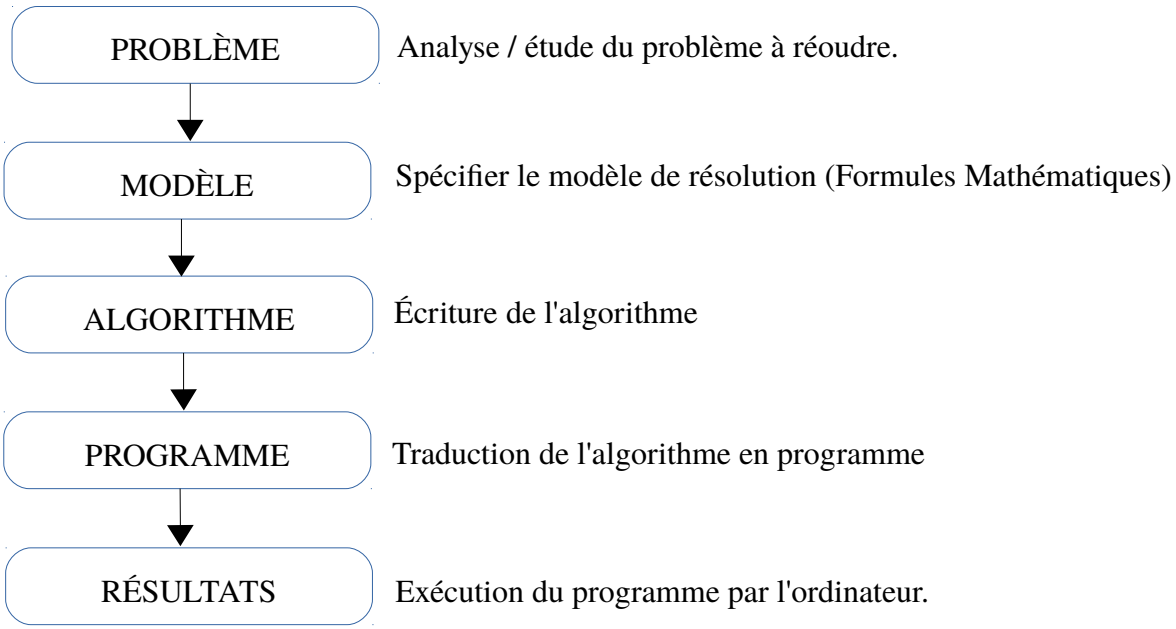
– Un algorithme est un ensemble d'*actions* (ou d'*instructions*) séquentielles et logiquement ordonnées, permettant de transformer des données en entrée (*Inputs*) en données de sorties (*outputs* ou les *résultats*), afin de résoudre un problème.

Donc, un algorithme représente une solution pour un problème donné. Cette solution est spécifiée à travers un ensembles d'instructions (séquentielles avec un ordre logique) qui manipulent des données. Une fois l'algorithme est écrit (avec n'importe quelle langues : français, anglais, arabe, *etc.*), il sera transformé, après avoir choisi un langage de programmation, en un programme code source qui sera compilé (traduit vers le langage machine : code binaire) et exécuté par l'ordinateur.

Pour le langage de programmation qui sera utilisé, ça sera le langage ***C***.

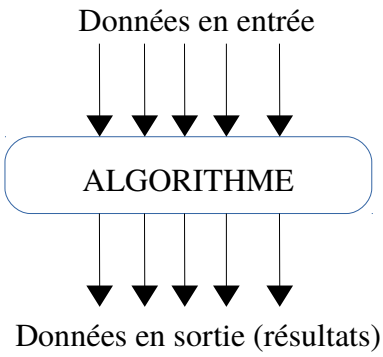
II.2. La démarche d'analyse d'un problème

Comme vu dans le point précédent, un algorithme représente une solution à un problème donné. Pour atteindre cette solution algorithmique un processus d'analyse et de résolution sera appliqué. Ce processus est constitué des étapes suivantes :



II.3. Données, Identificateurs, Types de données simples

Un algorithme permet de réaliser un traitement sur un ensemble de données en entrées pour produire des données en sorties. Les données en sorties représentent la solution du problème traité par l'algorithme. Un algorithme peut être schématisé comme suit :



Toute les données d'un programme sont des objets dans la mémoire vive (c'est un espace réservé

dans la RAM). Chaque objet (espace mémoire) est désigné par une appellation (*nom unique*) dite : *identificateur.*

II.3.1. Notion d'identificateur

Un identificateur est une chaîne de caractères contenant uniquement des caractère alpha-numériques (alphabétiques de [a-z] et [A-Z] et numérique [0-9]) et tiré 8 '_' (trait souligné), et qui doit commencer soit par un une lettre alphabétique ou _.

Un identificateur permet d'identifier d'une manière unique un algorithme (ou un programme), une variable, une constante, une procédure ou une fonction.

Dans un langage de programmation donnée, on a pas le droit d'utiliser les mots réservés (mots clés) du langage comme des identificateurs. Parmi les mots clés du langage C :

`auto, break, case, char, continue, do, double, else, extern, float, for, goto, if, int, long, return short, sizeof, static, struct, switch, typedef, union, unsigned, while`

Exemples :

`a1` : est un identificateur valide.
`a_1` : est un identificateur valide.
`A_1` : est un identificateur valide.
`x12y` : est un identificateur valide.
`x1 y` : est un identificateur non valide (à cause du blanc ou l'espace).
`x1-y` : est un identificateur non valide (à cause du signe -).
`x1_y` : est un identificateur valide.
1xy : est un identificateur non valide (commence un caractères numérique).

II.3.2. Constantes et variables

Les données manipulées par un algorithme (ou un programme) sont soit des constantes ou des variables :

– ***Constantes :*** une constante est un objet contenant une valeur qui ne peut jamais être modifiée. Son objectif est d'éviter d'utiliser une valeur d'une manière direct. Imaginons qu'un algorithme utilise la valeur *3.14* une dizaines de fois (le nombre d'occurrences de la valeur *3.14* est par exemple *15*) et qu'on veut modifier cette valeur par une autre valeur plus précise : *3.14159*. Dans ce cas on est amené à modifier toutes les occurrences de *3.14.* Par contre, si on utilise une constante

PI = 3.14 on modifier une seule fois cette constante.

- ***Variables :*** une variable est un objet contenant une valeur pouvant être modifiée.

Toute les données (variable ou constante) d'un algorithme possède un type de données (domaine de valeurs possibles).

II.3.3. Types de données

Dans l'algorithmique, nous avons cinq types de base :

- Entiers : représente l'ensemble {…, -4, -3, -2, -1, 0, 1, 2, 3, 4, ...}
- Réels : représente les valeurs numériques fractionnels et avec des virgule fixes (ou flottante)
- Caractères : représente touts les caractères imprimable.
- Chaînes de caractères : une séquence d'un ou plusieurs caractères
- Booléens (logique) : représente les deux valeurs *TRUE* et *FALSE*.

Le tableau suivant montre la correspondance entre les types d'algorithme et les types du langage de programmation C.

Algorithme	***C***
Entier	int
Réel	float
Booléen	*bool*
Caractère	char
chaîne	*Tableau de char (char[])*

Exemples de Déclarations de constantes :

```
CONST  PI = 3.14 ;                {constante réelle}
CONST  CHARGE = 1.6E-19;          {charge de l'électron}
CONST  MASSE = 0.9E-27;           {masse de l'électron}
CONST  message = 'Bonjour tous le monde';    {constante chaîne de caractère}
```

(Voir la page 9)

II.4. Structure d'un algorithme / programme

Un algorithme manipule des données, les données avant de les utiliser il faut les identifier et les

déclarer en utilisant les identificateur. Un algorithme est constitué de trois parties :

- ***Entête :*** dans - cette partie on déclare le nom de l'algorithme à travers un identificateur.
- ***Déclarations :*** dans cette partie on déclare toutes les données utilisées par l'algorithme.
- ***Corps :*** représente la séquence d'actions (instructions)

Pour écrire un algorithme, il faut suivre la structure suivante :

```
Algorithme <id_algorithme>
     <Déclarations>
Début
     <Corps : Instructions>
Fin.
```

```
#include <stdio.h>
int main()
{
     <déclarations>;
     <Instructions>;
     return 0;
}
```

Remarques

- Pour commenter un code écrit en programme C, on écrit les commentaires entre /* … */. Par exemple :

```
/* Ceci est un commentaire */
```

- On peut aussi utiliser les commentaire ligne :

```
// Ceci est un commentaire ligne
```

- En langage C, la partie déclaration peut être avant la fonction main, comme elle peut être dans la fonction main.
- En langage C, la fonction main est le point d'entrée du programme : la première qui sera exécutée.

II.4.1. Déclarations

Dans la partie déclaration, on déclare toutes les données d'entrées et de sorties sous forme de constantes et de variables.

- ***Constantes***

  Les constantes sont des objets contenant des valeurs non modifiables. Les constante sont déclarées comme suit :

  <identificateur> = <valeur>;

Exemples :

```
PI = 3.14;          {Constante réelle.}
MAX = 10;           {Constante entière.}
cc = 'a';           {Constante caractère.}
ss = "algo";        {Constante chaîne de caractère.}
b1 = true;          {Constante booléenne.}
b2 = false;         {Constante booléenne.}
```

En langage C

```
const float PI = 3.14;
const int MAX = 10;
const char cc = 'a';
const char ss[10] = "algo";
const int b1 = 0;
const int b2 = 1;
```

- ***Variables***

Les variables sont des objets contenant des valeurs pouvant êtes modifiées. Les variables sont déclarées comme suit :

<identificateur> : <type>;

Une variable appartient à un type de données. On a cinq types de données de base :

- *Entiers*
- *Réels*
- *Caractères*
- *Chaîne de caractères*
- *Booléens*, contenant deux valeurs : *True* ou *False* ;

Exemples :

```
x : réel                b1, b2, b3 : booleen
n, m : entier           s : chaîne
c1 : caractère
```

```
float x;                {variable réelle.}
int x,y;                {deux variables entières.}
char s[100];            {variable chaîne de caractères}
bool b1, b2, b3;        { #include <stdbool.h> }
char c1;                {variable caractère}.
```

N.B. : On peut, en plus des constantes et de variables, déclarer de nouveau types, des fonctions et procédures.

II.4.2. Corps

Le corps d'un algorithme est constitué d'un ensemble d'actions / instructions séquentiellement et logiquement ordonnées. Les instructions sont de cinq types, à savoir :

- ***Lecture*** **:** L'opération de faire entrer des données à l'algorithme. Une lecture consiste à donner une valeur arbitraire à une variable.
- ***Écriture*** **:** L'opération d'affichage de données. Elle sert à afficher des résultat ou des messages.
- ***Affectation*** **:** ça sert à modifier les valeurs des variables.
- ***Structure de contrôle*** **:** Elle permet modifier la séquentialité de l'algorithme, pour choisir un chemin d'exécution ou répéter un traitement.
  - *Structure de* ***Test alternatif simple / double***
  - *Structures* répétitives (itératives)
    - *la boucle* ***Pour***
    - *la boucle* ***Tant-que***
    - *la boucle* ***Répéter***

Dans le langage C, chaque instruction se termine par un *point-virgule*.

II.5. Types d'instructions

Toutes les instructions d'un programme sont écrites dans son corps. (entre *Début* et *Fin*, en C entre { et } de la fonction *main*). On peut regrouper ces instructions en deux grandes types :

- Les instructions séquentielles simples : Entrées, Sortie et Affectation
- Les structures de contrôles : Tests et boucles.

II.5.1. Instructions d'Entrées/Sorties (Lecture / Écriture)

II.5.1.1. Entrées (Lecture)

Une instruction d'entrée nous permet dans un programme de donner une valeur quelconque à une variable. Ceci se réalise à travers l'opération de lecture. La syntaxe et la sémantique d'une lecture est

comme suit :

Algorithme	***Langage C***	***Signification***
`Lire(<id_var>)`	`scanf("%c" ,<id_var>);`	Donner une valeur quelconque à la variable dont l'identifiant <*id_var*>.
`Lire(<iv1>, <iv2>, …);`	`scanf("%c %c %c", <iv1>, <iv2>, …);`	Donner des valeurs aux variables <iv1>, <iv2>, *etc.*

Il faut remarquer que l'instruction de lecture concerne uniquement les variables, on ne peut pas lire des constantes ou des valeurs (ou des expressions). Lors de la lecture d'une variable dans un programme C, le programme se bloque en attendant la saisie d'une valeur via le clavier. Une fois la valeur saisie, on valide par la touche ***entrée***, et le programme reprend l'exécution avec l'instruction suivante.

Exemples :

```
Lire (a, b, c)        scanf("%d %d %d", a, b, c);
lire (hauteur)        scanf("%f", hauteur);
```

II.5.1.1. Sorties (Écriture)

Une instruction de sortie nous permet dans un programme d'afficher un résultat (données traitées) ou bien un message (chaîne de caractères). Ceci se réalise à travers l'opération d'écriture. La syntaxe et la sémantique d'une écriture est comme suit :

Algorithme	***Langage C***	***Signification***
`Écrire(<id_var> | <id_const> | <valeur>, <expression>);`	`printf("%f", <id_var> | <id_const> | <valeur>, <expression>);`	Afficher une valeur d'une variable, d'une constante, valeur immédiate ou calculée à travers une expression.

Il faut remarquer que l'instruction d'écriture ne concerne pas uniquement les variables, on peut écrire des constantes, valeurs ou des expressions (arithmétiques ou logiques).

Exemples :

```
écrire('Bonjour')      printf ("Bonjour");          {afficher le message Bonjour}
écrire(a, b, c)        printf("%d %d %d "a, b, c);  {afficher les valeurs des
                                                    variables a, b et c}
écrire(5+2)                 printf("%f", 5+2);  {afficher le résultat de la somme de
                                                 5 et 2 : afficher 7}
écrire('La valeur de x : ', x);      printf("La valeur de x : %d", x);
```

II.5.2. Instruction d'affectation

Une affectation consiste à donner une valeur (immédiate, constante, variable ou calculée à travers une expression) à une variable. La syntaxe d'une affectation est :

```
En algorithmique : <id_varialbe> ← <valeur>|<id_variable>|<expression>;
En langage C     : <id_varialbe> =  <valeur>|<id_variable>|<expression>;
```

Une affectation possède deux parties : la partie gauche qui représente toujours une variable, et la partie droite qui peut être : une valeur fixe, constante, variable ou une expression. La condition pour qu'une affectation soit correcte est : la partie droite doit être du même type (ou de type compatible) avec la partie gauche.

Exemples :

`a ← 5;`	`a = 5;`	`{mettre la valeur 5 dans la variable a}`
`b ← a+5;`	`b = a+5;`	`{mettre la valeur de l'expression a+5 dans la variable B}`
`sup ← a>b;`	`sup = a>b;`	`{a>b donne un résultat booléen, donc sup est une variable booléenne}`

II.5.3. Structures de contrôles

En générale, les instructions d'un programme sont exécutés d'une manière séquentielle : la première instruction, ensuite la deuxième, après la troisième et ainsi de suite. Cependant, dans quelques cas, on est amené soit à choisir entre deux ou plusieurs chemins d'exécution (un choix entre deux ou plusieurs options), ou bien à répéter l'exécution d'un ensemble d'instructions, pour cela nous avons besoins des structures de contrôle pour contrôler et choisir les chemins (le flux) d'exécutions ou refaire un traitement plusieurs fois. Les structures de contrôle sont de deux types : Structures de contrôles conditionnelles (Tests) et structures de contrôle répétitives (itératives) (Boucles).

II.5.3.1. Structures de contrôle conditionnelle

Ces structures sont utilisées pour décider de l'exécution d'un bloc d'instruction : est-ce-que ce bloc est exécuté ou non. Ou bien pour choisir entre l'exécution de deux blocs différents. Nous avons deux types de structures conditionnelles :

a. Test alternatif simple

Un test simple contient un seul bloc d'instructions. Selon une condition (expression logique), on décide est-ce-que le bloc d'instructions est exécuté ou non. Si la condition est vraie, on exécute le bloc, sinon on l'exécute pas. La syntaxe d'un test alternatif simple est comme suit :

```
si <Condition> alors
    <bloc_instructions_si>;
finsi;
```

```
if (<condition>)
{
    <bloc_instructions_if>;
}
```

Exemple :

```
lire(x);
si x > 2 alors
    x ← x + 3;
finsi
écrire (x)
```

```
scanf("%d", x);
if (x > 2)
{
    x = x + 3;
}
printf("%d", x);
```

Remarque : Dans le langage C, un bloc est délimité par deux accolades **{** et **}**. Si le bloc contient une seule instruction, les accolades sont **{** et **}** sont facultatifs (on peut les enlever).

b. Test alternatif double

Un test double contient deux blocs d'instructions : on est amené à décider entre le premier bloc ou le seconds. Cette décision est réalisée selon une condition (expression logique ou booléenne) qui peut être vraie ou fausse. Si la condition est vraie on exécute le premier bloc, sinon on exécute le second.

La syntaxe d'un test alternatif simple est :

```
si <Condition> alors
    <bloc_inst_si>;
sinon
    <bloc_inst_sinon>;
finsi
```

```
if (<condition>)
{
    <bloc_inst_if>;
}
else
{
    <bloc_inst_else>;
}
```

Exemple :

```
lire(x)
si x > 2 alors
        x ← x + 3
sinon
        x ← x - 2
finsi
écrire (x)
```

```
scanf("%d", x);
if (x > 2){
        x:= x + 3;
}
else {
        x:= x - 2;
}
printf("%d", x);
```

Remarques :

- Dans le langage C, il ne faut pas point-virgule après la condition (erreur logique)
- Dans l'exemple précédent, on peut enlever **{ }** du **if** et ceux du **else** puisqu'il y a une seule instruction dans les deux blocs.

Exemples :

1- Écrire un algorithme (et un programme C) qui permet d'indiquer si un nombre entier est pair ou non.

II.5.3.2. Structures de contrôle répétitives

Les structures répétitives nous permettent de répéter un traitement un nombre fini de fois. Par exemple, on veut afficher tous les nombre premier entre 1 et N (N nombre entier positif donné). Nous avons trois types de structures itératives (boucles) :

a. Boucle Pour (For)

La structure de contrôle répétitive **pour** (**for** en langage C) utilise un indice entier qui varie (avec un incrément = 1) d'une valeur initiale jusqu'à une valeur finale. À la fin de chaque itération, l'indice est incrémenté de **1** d'une manière automatique (implicite).

La syntaxe de la boucle **pour** est comme suit :

```
pour <ind>←<vi> à <vf> faire
        <instruction(s)>
finPour;
```

```
for (<ind>:=<vi>;<ind> <= <vf>;<ind>++)
 {
        <instruction(s)>;
 }
```

<indice> : variable entière (compteur de la boucle pour)
<vi> : valeur initiale <vf> : valeur finale

La boucle pour contient un bloc d'instructions (les instructions à répéter). Si le bloc contient une seule instruction, les accolades **{** et **}** sont facultatifs.

Le bloc sera répété un nombre de fois = (<vf> - <vi> + 1) si la valeur finale est supérieure ou égale à la valeur initiale. Le bloc sera exécuté pour <indice> = <vi>, pour <indice> = <vi>+1, pour <indice> = <vi>+2, …, pour <indice> = <vf>.

c. Boucle Tant-que (while)

La structure de contrôle répétitive **tant-que** (**while** en langage C) utilise une expression logique (booléenne) comme condition d'accès à la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) donc on entre à la boucle, sinon on quitte la boucle.

La syntaxe de la boucle **tant-que** est comme suit :

```
tant-que <condition> faire
    <instruction(s)>
finTant-que;
```

```
while (<condition>)
 {
     <instruction(s)>;
 }
```

<condition> : expression logique qui peut être vraie ou fausse.

On exécute le bloc d'instructions tant que la condition est vraie. Une fois la condition devinne fausse, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après fin *Tant que* (après **}**).

Toute boucle **pour** peut être remplacée par une boucle **tant-que**, cependant l'inverse n'est pas toujours possible.

c. Boucle Répéter (Repeat)

La structure de contrôle répétitive **répéter** utilise une expression logique ou booléenne comme condition de sortie de la boucle : si la condition est vérifiée (elle donne un résultat vrai : TRUE) on sort de la boucle, sinon on y accède (on répète l'exécution du bloc).

La syntaxe de la boucle **répéter** est comme suit :

```
repéter
    <instruction(s)>
jusqu'à <condition>;
```

```
do {
     <instruction(s)>;
while (<condition2>);
```

<condition> : expression logique qui peut être vraie ou fausse.

<condition2> : expression logique inverse de <condition>.

On exécute le bloc d'instructions jusqu'à avoir la condition correcte. Une fois la condition est vérifiée, on arrête la boucle, et on continue l'exécution de l'instruction qui vient après **jusqu'à**. Dans la boucle **do** … **while** (en langage C) la condition est l'inverse de la condition de Répéter.

La différence entre la boucle **répéter** et la boucle **tant-que** est :

- La condition de **répéter** et toujours l'inverse de la condition **tant-que** : pour **répéter** c'est la condition de sortie de la boucle, et pour **tant-que** c'est la condition d'entrer.
- Le teste de la condition est à la fin de la boucle (la fin de l'itération) pour **répéter**. Par contre, il est au début de l'itération pour la boucle **tant-que**. C'est-à-dire, dans **tant-que** on teste la condition avant d'entrer à l'itération, et dans **répéter** on fait l'itération après on teste la condition.

II.6. Représentation en organigramme

Un organigramme est la représentation graphique de la résolution d'un problème. Il est similaire à un algorithme. Chaque type d'action dans l'algorithme possède une représentation dans l'organigramme.

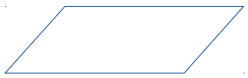
Il est préférable d'utiliser la représentation algorithmique que la représentation par organigramme notamment lorsque le problème est complexe.

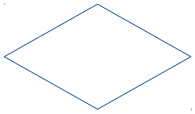
Les inconvénients qu'on peut rencontrer lors de l'utilisation des organigrammes sont :

- Quand l'organigramme est long et tient sur plus d'une page,
- problème de chevauchement des flèches,
- plus difficile à lire et à comprendre qu'un algorithme.

II.6.1. Les symboles d'organigramme

Les symboles utilisés dans les organigrammes :

	Représente le début et la Fin de l'organigramme
	Entrées / Sorties : Lecture des données et écriture des résultats.
	Calculs, Traitements

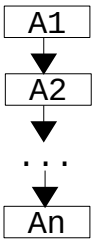
	Tests et décision : on écrit le test à l'intérieur du losange
	Ordre d'exécution des opérations (Enchaînement)
	Connecteur

II.7.2. Représentation des primitives algorithmiques

II.7.2.1. L'enchaînement

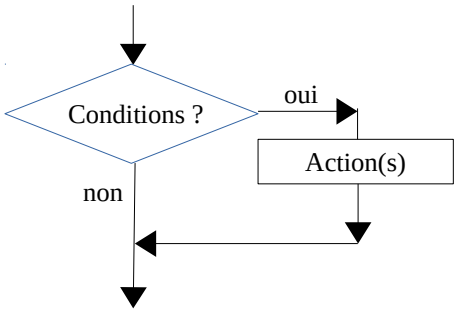
L'enchaînement permet d'exécuter une série d'actions dans l'ordre le leur apparition. Soit A_1, A_2, …,A_n une série d'actions, leur enchaînement est représenté comme suit :

```
A1;
A2;
 .
 .
An;
```



A1, A2, …, An : peuvent être des actions élémentaires ou complexes.

II.7.2.2. La structure alternative simple

Représentation algorithmique	***Représentation sous forme d'organigramme***
si <Condition> **alors** <action(s)>; **finsi**; Si la condition est vérifiée, le bloc <*action(s)*> sera exécuté, sinon rien, et on continue l'exécution de l'instruction après fin si.	

Les condition utilisées pour les teste (simple ou double) sont des expression logique ou booléennes, ça veut dire des expression dont leur évaluation donne soit TRUE (Vrai) ou FALSE (faux). Toute comparaison entre deux nombre représente un expression logique. On peut former des expressions logiques à partir d'autres expressions logique en utilisant les opérateurs suivant : Not, Or et And.

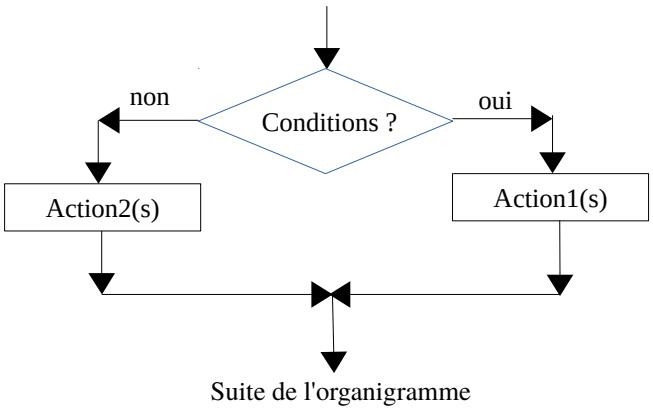
Exemples :

(x >= 5) : est une expression logique, elle est vrai si la valeur de x est supérieur ou égale à 5. elle est fausse dans le cas contraire.

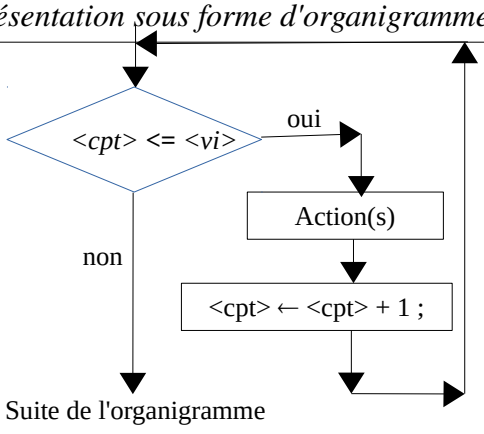
Not (x >= 5) : E.L. qui est vrai uniquement si la valeur de x est inférieur à 5.

(x >=5) And (y<=0) : E.L. qui est vrai si x est supérieur ou égale à 5 et y inférieur ou égale à 0.

II.7.2.3. La structure alternative double

Représentation algorithmique	*Représentation sous forme d'organigramme*
si <Condition> **alors** <action1(s)>; **sinon** <action2(s)>; **finsi**; Si la condition est vérifiée, le bloc <*action1(s)*> sera exécuté, sinon (si elle est fausse) on exécute <*action2(s)*>.	

II.7.2.4. La structure itérative POUR (Boucle POUR)

Représentation algorithmique	*Représentation sous forme d'organigramme*
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour**;	

Dans la boucle **POUR**, on exécute le bloc <*acitons*> *(<vf> - <vi> + 1)* fois. Ceci dans le cas où <*vf*> est supérieur ou égale à <*vi*>. Dans le cas contraire, le bloc d'actions ne sera jamais exécuté. Le déroulement de la boucle POUR est exprimé comme suit :

1 – la variable entière <*cpt*> (le compteur) prends la valeur initiale <*vi*> ;

2 – on compare la valeur de <*cpt*> à celle de <*vf*> ; si <*cpt*> est supérieur à <*vf*> on sort de la boucle ;

3 – si <*cpt*> est inférieur ou égale à <*vf*> on exécute le bloc <*action(s)*> ;

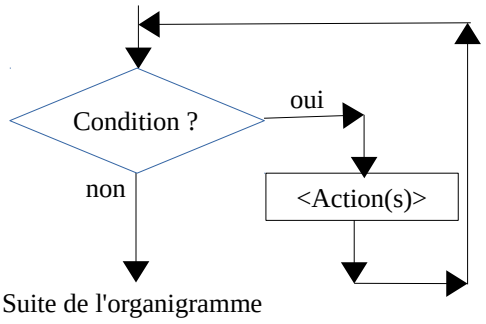
4 – la boucle *POUR* incrémente automatiquement le compteur <*cpt*>, c'est-à-dire elle lui ajoute *un* (<cpt> ← <cpt> + 1);

5 – on revient à 2 (pour refaire le teste <*cpt*> **<=** <*vi*> C'est pour cela qu'on dit la boucle);

Remarque :

La boucle **POUR** est souvent utilisée pour les structures de données itératives (les tableaux et les matrices – variables indicées).

II.7.2.5. La structure itérative Tant-Que (Boucle Tant-Que)

Représentation algorithmique	*Représentation sous forme d'organigramme*
Tant-que <condition> **faire** <action(s)>; **finpour**;	

On exécute le bloc d'instructions <actions> tant que la <*condition*> est vérifiée (c'est-à-dire elle est vraie). Le déroulement de la boucle est comme suit :

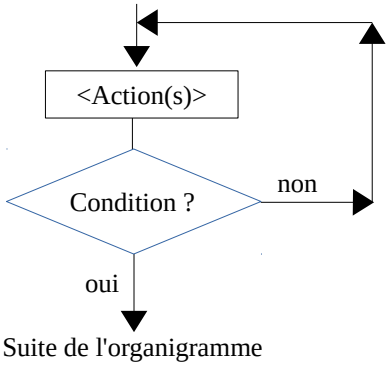
1 – On évalue la condition : si la condition est fausse on sort de la boucle ;

2 – Si la condition est vraie, on exécute le bloc <actions> ; sinon va à **4**.

3 – On revient à 1 ;

4 – On continue la suite de l'algorithme

II.7.2.6. La structure itérative Répéter (Boucle Répéter)

Représentation algorithmique	*Représentation sous forme d'organigramme*
Répéter <action(s)>; **Jusqu'à** <condition>;	

On répète l'exécution du bloc <*action(s)*> jusqu'à avoir la condition correcte. Le déroulement est comment suit :

1 – On exécute le bloc <*action(s)*> ;

2 – On évalue la condition : si la condition est vérifiée (elle est vraie) on sort de la boucle (on continue la suite de l'algorithme);

3- si la condition n'est pas vérifiée (elle est fausse) on revient à 1.

Remarques :

✔ N'importe quelle boucle ***POUR*** peut être remplacée par une boucle ***Tant-Que***, cependant l'inverse n'est pas toujours correcte, c'est-à-dire, il y a des cas où la boucle ***Tant-Que*** ne peut pas être remplacée par une boucle ***POUR***.

✔ On transforme une boucle pour à une boucle Tant-Que comme suit :

*Boucle **POUR***	*Boucle **Tant-Que***
pour <cpt> ← <vi> **à** <vf> **faire** <action(s)>; **finpour;**	<cpt> ← <vi>; **Tant-que** <cpt> <= <vf> **faire** <action(s)>; <cpt> ← <cpt> + 1; **finTant-Que**;

✔ La boucle Répéter possède une condition de sortie (c'est-à-dire si elle est vraie on sort de la boucle), alors que la boucle Tant-que possède une condition d'entrée (c'est-à-dire si elle est vraie on entre dans la boucle).

✔ La boucle Répéter exécute le bloc <*action(s)*> au moins une fois, le teste vient après l'exécution du bloc.

✔ La boucle Tant-Que peut ne pas exécuter le bloc <action(s)> (dans le cas ou la condition est fausse dès le début), puisque le teste est avant l'exécution du bloc.

II.7. Exemples d'Application

II.7.1. Exemple 1 : Permutation de valeurs de deux variables

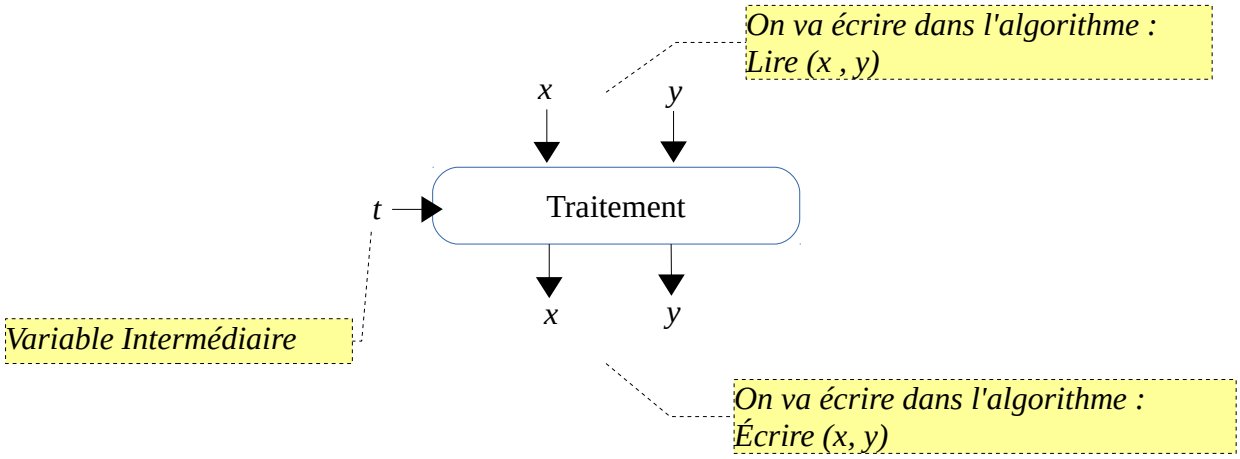
Écrire un algorithme qui permet de permuter les valeurs de deux variables réelles. Par exemple $x = 18$ et $y = -20$ au début de l'algorithme, à la fin on obtient $x = -20$ et $y = 18$.

Traduire l'algorithme en programme C, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : On veut réaliser la permutation de deux variables, on doit tout d'abord donner deux valeurs aux deux variables (on aura besoins de deux lectures). À la fin, on doit afficher les mêmes variables, après avoir été permutées.

Donc L'algorithme possède deux variables d'entrée et deux variables de sortie. Les variables de sortie sont les mêmes variables d'entrée. On peut schématiser l'algorithme comme suit :



Après avoir déterminé les variables d'entrée et de sortie, il nous reste à trouver l'idée du

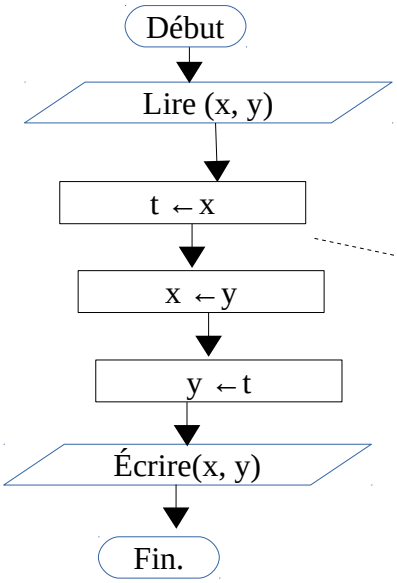
traitement. La solution qui vient immédiatement à l'esprit est : $x \leftarrow y$ et $y \leftarrow x$. (On affecte y à x, enduite on affecte x à y). Le problème de cette solution est qu'on va perdre la valeur initiale de x. (Explication : …). il faut conserver tout d'abord la valeur de x dans une autre variable, par exemple t, en suite, on affecte t à y. (t $\leftarrow x$ et $x \leftarrow y$ enfin $y \leftarrow t$).

Donc l'algorithme (et sa traduction en programme C) sera comme suit :

```
Algorithme exemple1;
      Variables
            x, y, t : reel;
Début
      Lire (x, y)
      t ← x
      x ← y
      y ← t
      Ecrire (x, y)
Fin
```

```
#include <stdio.h>;

int main(){
      float x, y, t;
      scanf('Donnez la valeur de x et y : ');
      scanf ("%f %f" , &x, &y);
      t = x;
      x = y;
      y = t;
      printf("x = %f et y = %f ", x, y);
}
```



- Dans l'organigramme, on schématise le chemin d'exécution, et on indique le séquencement des actions à réaliser. On remarque dans l'organigramme à gauche, qu'il y a un seul chemin d'exécution (un seul sens pour les flêches). Dans ce cas l'exécution est déterministe (On connais au préalables toutes les actions qui seront exécutée).

- Dans les exemples qui suivent, on peut éventuellement trouver plusieurs chemins d'exécution.

N.B. : On peut généraliser le problème, en permutant, d'une manière circulaire, les valeurs de trois variable x, y et z. (aussi pour 4, 5, et ainsi de suite de variables).

Le déroulement : On déroule l'algorithme pour $x = 6.5$ et $y = 17$

Instructions	***Variables***		
	x	*y*	*t*
Lire (x, y)	*6.5*	*17*	*/*
t ← x	*"*	*"*	*6.5*
x ←y	*17*	*"*	*"*

y ←t	*17*	*6.5*	*''*
Écrire (x,y)	*17*	*6.5*	*''*

II.7.2. Exemple 2 : Somme de deux variables

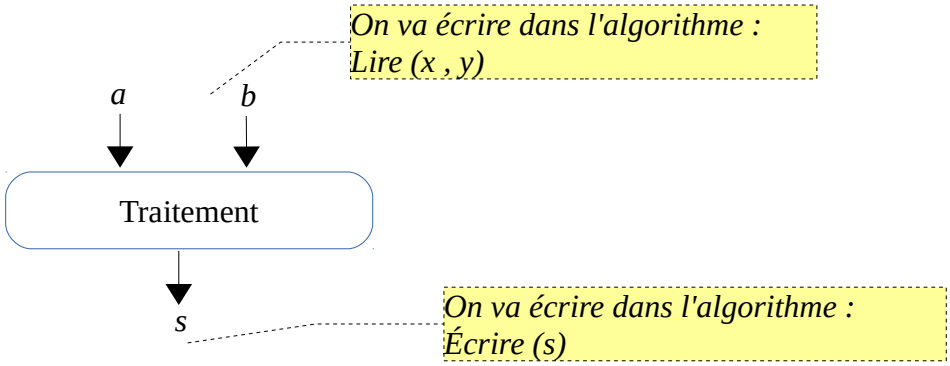
Écrire un algorithme qui permet de réaliser la somme de deux variables entières.

Traduire l'algorithme en programme C, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : On veut réaliser la somme de deux variables entière *a* et *b*, on doit tout d'abord donner deux valeurs aux deux variables (on aura besoins de deux lectures). À la fin, on doit calculer leur somme dans une troisième variable *s* et afficher ensuite la valeur de *s*.

Donc L'algorithme possède deux variables d'entrée (*a* et *b*) et une variable de sortie *s*. On peut schématiser l'algorithme comme suit :



Après avoir déterminé les variables d'entrée et de sortie, il nous reste à trouver l'idée du traitement. Le traitement est simple, il suffit de réaliser l'affectation suivante : $s \leftarrow a + b$.

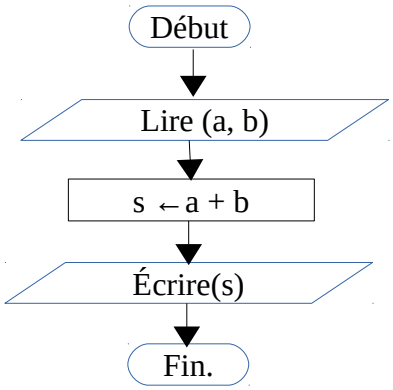
Donc l'algorithme (et sa traduction en programme C) sera comme suit :

```
Algorithme somme;
      Variables
            a, b, s : entier
Début
      Lire (a, b)
      s ← a + b
      Écrire (s)
Fin
```

```c
#include <stdio.h>

int main(){
      float x, y, t;
      printf("Donnez la valeur de x et y : ");
      scanf ("%f %f" , &x, &y);
      t = x;
      x = y;
      y = t;
      printf("x = %f et y = %f ", x, y);
}
```

Vous pouvez pouvez exécuter le programme en ligne : https://onlinegdb.com/s3WtqF4y1



Le déroulement : On déroule l'algorithme pour $a = 5$ et $y = -16$

Instructions	***Variables***		
	a	***b***	***s***
Lire (a,b)	*5*	*-16*	*/*
s ← a+b	*"*	*"*	*-11*
écrire(S)			*-11*

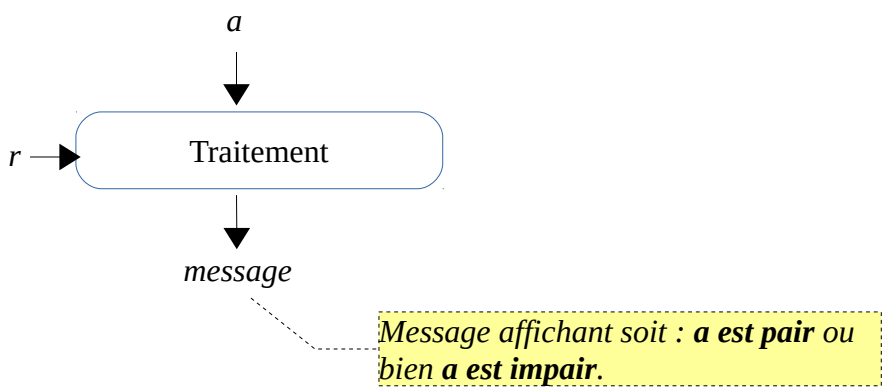
II.7.3. Exemple 3 : Nombre pair ou impair

Écrire un algorithme qui permet d'indiquer si le nombre est pair ou non (un nombre pair est divisible par 2).

Traduire l'algorithme en programme C, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : L'algorithme prend comme donnée d'entrée une variable entière et affiche un message indiquant si ce nombre là est pair ou non. Un nombre pair est divisible par 2. Donc, nous avons une variable d'entrée et un message comme sortie



Pour savoir si un nombre est pair ou non, il suffit de calculer son reste de division par 2. On peut utiliser directement la fonction *mod* qui s'écrit : *n mod b* = le reste de division de *n* sur *b*.

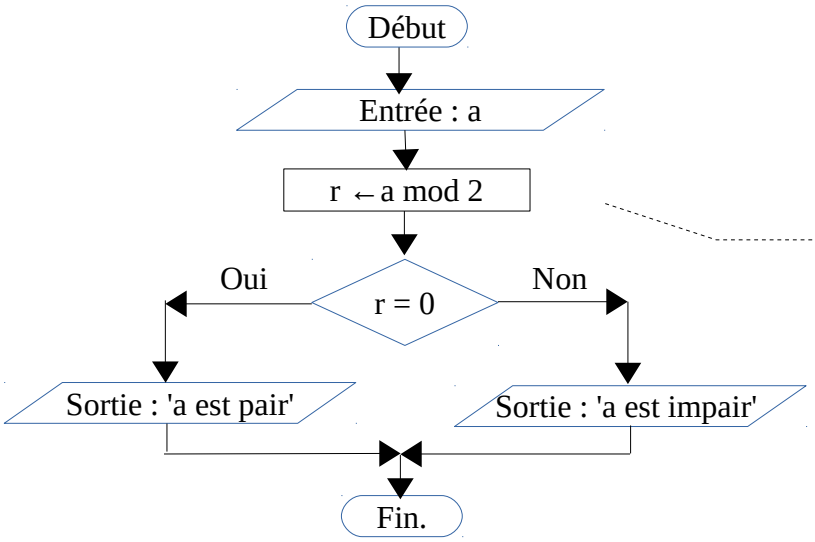
Donc l'algorithme (et sa traduction en programme C) sera comme suit :

```
Algorithme pair_impair
      Variables
            a, r : entier
Début
      Lire (a)
      r ← a mod 2
      si (r = 0) alors
         écrire ('a est pair')
      sinon
         écrire ('a est impair')
      finsi
Fin
```

```
#include <stdio.h>

int main(){
      int a, r;
      printf("Donnez une valeur entière : ");
      scanf ("%d" , &a);
      r = a % 2;
      if (r == 0)
            printf("a est pair ");
      else
            printf("a est impair ");
}
```

Vous pouvez pouvez exécuter le programme en ligne : *https://onlinegdb.com/EMpEAry6f*



- Dans cet exemple, on remarque dans l'organigramme que nous avons deux chemins d'exécution. Pour cela, on peut avoir deux scénario de déroulement de l'algorithme : le premier si a est pair et le second si a est impair.

Le déroulement : On déroule l'algorithme pour $a = 7$

Instructions	***Variables***	
	a	***r***
Lire (a)	*7*	*/*
r ← a mod 2	*"*	*1 (= 7 mod 2)*
R = 0 → FALSE *On suit la branche NON* *→ écrire ('a est impair')*	*a est impair*	

On déroule l'algorithme pour $a = 16$

Instructions	***Variables***	
	a	***r***
Lire (a)	*16*	*/*
r ← a mod 2	*''*	*0 (= 16 mod 2)*
R = 0 → TRUE *On suit la branche OUI* *→ écrire ('a est pair')*	*a est pair*	

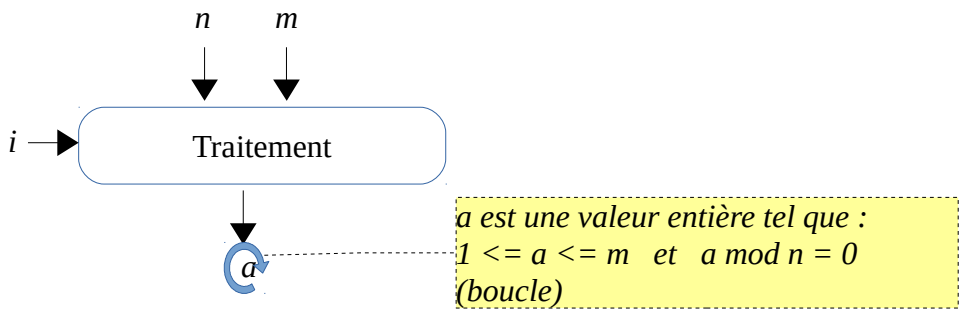
II.7.4. Exemple 4 : Afficher tous les nombres divisibles par n

Écrire un algorithme qui permet d'afficher tous les nombres entiers supérieurs ou égale à *1* et inférieurs à *m* (entier positif) et et qui sont divisible sur un nombre entier *n*.

Traduire l'algorithme en programme C, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : Pour chercher tous les nombres qui sont entre *1* et *m*, on doit donner une valeur pour *m* (donc *m* est une variable d'entrée). Et si on indique que ces nombres sont divisibles par *n*, on doit aussi donner une valeur pour *n* (donc *n* est aussi une variable d'entrée). Comme résultat, l'algorithme affiche tous les nombres entiers divisibles par *n* et qui sont entre *1* et *m*.



Pour le traitement, on parcours tous les nombres *a* entre *1* et *m* et à chaque itération (boucle) on teste si *a* est divisible par *n* ou non. Dans le cas ou *a* est divisible on l'affiche sinon on fait rien.

Donc l'algorithme (et sa traduction en programme C) sera comme suit :

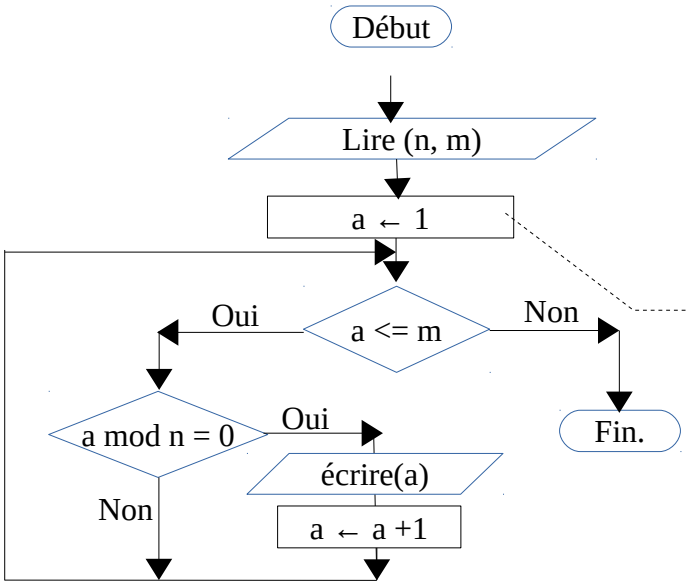
```
Algorithme exemple_4
      Variables
            n,m,a : entier
Début
      Lire (n, m)
      pour a ← 1 à m faire
         si (a mod n = 0) alors
            écrire (a)
         finsi
      finPour
Fin
```

```
#include <stdio.h>

int main(){
      int n, m, a;
      printf("Donnez deux entier n et m : ");
      scanf ("%d %d" , &n, &m);

      for (a=1 ; a<=m ; a++) {
         if (a % n == 0){
             printf("%d \n", a);
         }
      }
}
```

Vous pouvez pouvez exécuter le programme en ligne : https://onlinegdb.com/u-PY5Ultu



*- La boucle **pour** initialise le compteur (a←1), réalise le teste (a <= m) et incrémente le compte de 1 (a←a+1) à la fin de chaque itération.*

- L'organigramme montre le chemin de la boucle sous forme d'un circuit bouclé.

*- Si le teste a<=m est faux, on sort de la boucle **pour** (et dans cet exemple, on quitte l'algorithme).*

Le déroulement : On déroule l'algorithme pour $n = 3$ et $m = 7$

Instructions	***Variables***		
	n	***m***	***a***
Lire (n, m)	*3*	*7*	*/*
Pour a = 1 *a mod n = 0* *(1 mod 3 = 0) → **false*** *donc on fait rien*	*''*	*''*	*1*
Pour a = 2 *a mod n = 0* *(2 mod 3 = 0) → **false*** *donc on fait rien*	*''*	*''*	*2*

Pour a = 3 *a mod n = 0* *(3 mod 3 = 0) → **true*** *donc on affiche a = 3*	''	''	*3*
Pour a = 4 *a mod n = 0* *(4 mod 3 = 0) → **false*** *donc on fait rien*	''	''	*4*
Pour a = 5 *a mod n = 0* *(5 mod 3 = 0) → **false*** *donc on fait rien*	''	''	*5*
Pour a = 6 *a mod n = 0* *(6 mod 3 = 0) → **true*** *donc on affiche a = 6*	''	''	*6*
Pour a = 7 *a mod n = 0* *(7 mod 3 = 0) → **false*** *donc on affiche a = 3*	''	''	*7*
Pour a = 8 *on arrête la boucle* *(a>m 8 > 7)* *fin de l'algorithme*	''	''	*8*

N.B. :

– La boucle **Pour** initialise le compteur (c'est une variable entière) de la valeur initiale une seule fois (le premier passage).

– À la fin de chaque itération, on ajoute *1* au compteur de la boucle **Pour**.

– Avant d'accéder à une itération, on réalise une comparaison entre le compteur et la valeur finale (est-ce-que le compteur est inférieur ou égale à la valeur finale. Si oui, on accède à la boucle,

sinon on arrête la boucle).

- Si la valeur finale est strictement inférieure à la valeur initiale, dans ce cas, on accède jamais à la boucle **Pour**.
- On peut remplacer la boucle **Pour** soit par la boucle **Tant-Que** ou bien la boucle **Répéter**.

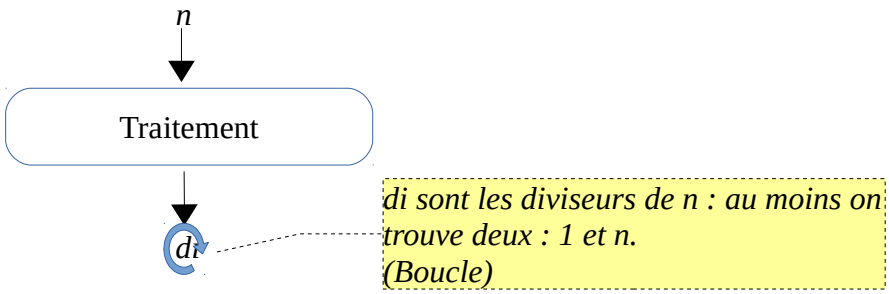
II.7.5. Exemple 5 : *Rechercher les diviseurs d'un nombre n*

Écrire un algorithme qui permet d'afficher tous les diviseurs positifs d'un nombre entier *n*.

Traduire l'algorithme en programme C, et réaliser le déroulement de cette algorithme.

Solution

Analyse et Discussion : Pour trouver tous les diviseurs d'un nombre entier n, il suffit de parcourir les nombre de *1* jusqu'à *n*. Mais, il faut faire attention dans le cas où *n=0*. Dans ce cas tous les nombres entiers sont des diviseurs de *n*.



Pour le traitement, on parcours tous les nombres *di* entre *1* et *n* et à chaque itération (boucle) on teste si *n* est divisible par *di* ou non. Dans le cas ou *n* est divisible on affiche alors *di* sinon on fait rien. Donc l'algorithme (et sa traduction en programme C) sera comme suit :

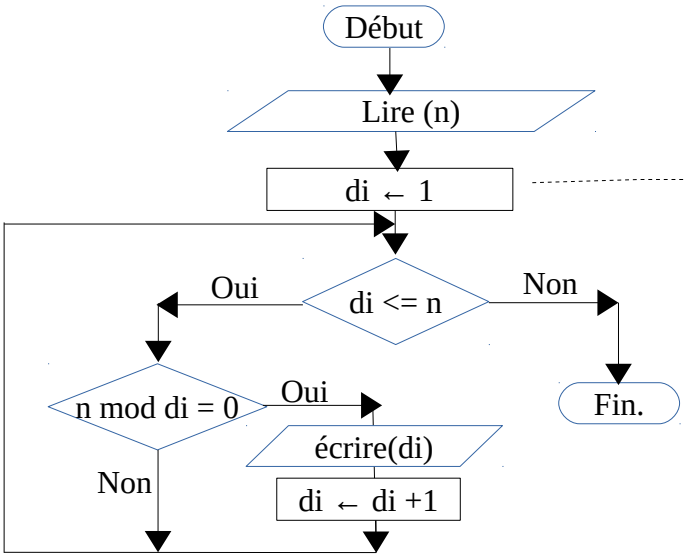
```
Algorithme exemple_5
      Variables
            n,di : entier
Début
      Lire (n)
      pour di ← 1 à n faire
         si (n mod di = 0) alors
            écrire (di)
         finsi
      finPour
Fin
```

```
#include <stdio.h>

int main(){
      int n, di;
      printf("Donnez la valeur de n : ");
      scanf ("%d" , &n);

      for (di=1 ; di<=n ; di++) {
         if (n % di == 0){
             printf("%d \n", di);
         }
      }
}
```

Vous pouvez pouvez exécuter le programme en ligne : https://onlinegdb.com/u-PY5Ultu



*- Dans cette organigramme (algorithme), on a pas traiter le cas de n = 0 (**Donc, essayer de le faire**)*

- Aussi, il faut traiter le cas de n < 0.

Le déroulement : On déroule l'algorithme pour $n = 12$

Instructions	***Variables***	
	n	***di***
Lire (n)	*12*	*/*
Pour di = 1 *n mod di = 0* *(12 mod 1 = 0) → **true*** *donc on affiche di = 1*	*''*	*1*
Pour di = 2 *n mod di = 0* *(12 mod 2 = 0) → **true*** *donc on affiche di = 2*	*''*	*2*
Pour di = 3 *n mod di = 0* *(12 mod 3 = 0) → **true*** *donc on affiche di = 3*	*''*	*3*
Pour di = 4 *n mod di = 0* *(12 mod 4 = 0) → **true*** *donc on affiche di = 4*	*''*	*4*
Pour di = 5	*''*	*5*

n mod di = 0 *(12 mod 5 = 0) → **false*** *donc on fait rien*		
Pour di = 6 *n mod di = 0* *(12 mod 6 = 0) → **true*** *donc on affiche di = 6*	*''*	*6*
Pour di = 7 *n mod di = 0* *(12 mod 7 = 0) → **false*** *donc on fait rien*	*''*	*7*
Pour di = 8 *n mod di = 0* *(12 mod 8 = 0) → **false*** *donc on fait rien*	*''*	*8*
Pour di = 9 *n mod di = 0* *(12 mod 9 = 0) → **false*** *donc on fait rien*	*''*	*9*
Pour di = 10 *n mod di = 0* *(12 mod 10 = 0) → **false*** *donc on fait rien*	*''*	*10*
Pour di = 11 *n mod di = 0* *(12 mod 11 = 0) → **false*** *donc on fait rien*	*''*	*11*
Pour di = 12 *n mod di = 0* *(12 mod 12 = 0) → **true*** *donc on affiche di = 12*		*12*

Donc, l'algorithme affichera pour les diviseurs de *12 : 1, 2, 3, 4, 6, 12*

Bon Courage
&
Travaillez Bien.