I Définitions

Définition 1

Un algorithme est ...

Définition 2

Un programme est ...

Corollaire 0.1 : Un problème peut posséder ...

e.g. un plat possède plusieurs recettes de cuisine et beaucoup de personnes savent le préparer.

Étape de résolution :

* comprendre le problème
* décomposer en sous-problème simple à résoudre
* associer à chaque sous-problème :
- les données nécessaires
- les données résultantes
- préciser la démarche à suivre pour arriver au résultat en partant des données

Appliquer pour chacun des exemples ci-dessous, les étapes de résolution.

expl1 : vérifier que 2 fractions sont égales a/b et a'/b'

...

expl2 : vérifier qu'un triangle est droit

...

⚠ Il faut prendre soin de préciser les ..... de son programme.

Afin de vérifier la qualité d'un programme, à défaut de démontrer sa correction (c'est parfois possible *cf.* chapitre sur les tris), on va chercher à le tester avec des .....

II Organigramme et programmation

Une variable est une sorte de boîte portant un nom (le nom de la variable) et contenant un objet. Ainsi,

Définition 3

Une variable est ...

Rq : en `python`, une variable est formé d'un seul mot commençant nécessairement par une lettre. On évitera les accents et caractères spéciaux (sauf l'underscore).

1 typage

Souvent en programmation, un objet possède ..... qui peut être un entier, un caractère, un nombre, une liste de caractères, une liste de nombres, une image, etc. On dit alors que la variable est de ce type.

Dans les langages de programmations typés, les variables sont typées et on ne peut généralement pas faire des opérations entre deux types différents.

Le type d'une variable est souvent défini en début d'algorithme ou au début d'un programme à typage statique. Dans un langage de programmation à typage dynamique (*e.g.* `python`), le typage n'est pas définit par l'utilisateur mais se fait en cours d'éxécution.

expl3 : en `python`,

```
type(1)
type(1.1)
type( "nsi" )
```

`type(1)` renvoie ...
`type(1.)` renvoie ...
`type("nsi")` renvoie ...

Rq : une chaîne de caractères est toujours notée entre guillemets (' ou "), c'est ce qui la diffère des variables.

2 affectation

L'affectation se fait simplement par un signe égal (=). La valeur de gauche se voit attribuer la valeur de droite ; ainsi = correspond à la flèche ←.

expl4 :

algorithme	**organigramme**	**python**
integer : x x **PREND LA VALEUR** 3	$x \leftarrow 3$	`x=3`

expl5 :

algorithme	**organigramme**	**python**
...	$y \leftarrow 7{,}45$	...

les calculs ne sont pas toujours exacts. Nous verrons au chapitre des représentations des flottants pourquoi certaines valeurs de nombres sont approchées, et ainsi ces flottants peuvent être considérés comme des pseudo-réels. Pour l'instant, remarquons en `python`,

```
from math import sqrt
sqrt(2)**2
```

$\left(\sqrt{2}\right)^2$ renvoie 2,000 000 000 000 000 4 (!)

Rq : en `python` la virgule numérique se note par un '.'

? Éxiste-t-il des types compatibles (ou semi-compatibles) pour certaines opérations ?
Lorsque le tout englobe la partie (les integer sont compris dans les float ; les character sont compris dans les string), les types sont généralement compatibles. Ainsi integer / float ; character / string sont compatibles.

expl6 : En `python`,

```
type( 3+7.1)
3*"ah"
```

$3 + 7.1$ est de type ...
`3*"ah"` renvoie ...

3 séquence d'instructions

Nous avons vu qu'un algorithme est une séquence d'instructions. L'ordre a-t-il vraiment une importance ?

Sur ces deux exemples, on cherche à échanger la valeur de deux variables entre elles.

algorithme	**organigramme**	**python**
integer : x, y x **PREND LA VALEUR** 3 y **PREND LA VALEUR** 5 y **PREND LA VALEUR** x x **PREND LA VALEUR** y	$x \leftarrow 3$ $y \leftarrow 5$ $y \leftarrow x$ $x \leftarrow y$	... ...

Au final, $x = \ldots$ et $y = \ldots$.

algorithme

integer : x, y
x **PREND LA VALEUR** 3
y **PREND LA VALEUR** 5
x **PREND LA VALEUR** y
y **PREND LA VALEUR** x

organigramme

python

```
...
...
...
...
```

Au final, $x = \ldots$ et $y = \ldots$.

Nota : dans ces organigrammes afin de gagner de la place la première et dernière instruction sont représentées dans une ellipse. Les affectations le sont dans des rectangles.

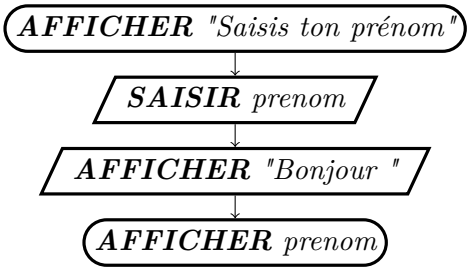
4 entrée/sortie

Afin de communiquer avec l'extérieur, les algorithmes et programmes permettent des ..... Sur ordinateur, ces entrées se font généralement au travers ...

algorithme

string : prenom
AFFICHER "Saisis ton prénom"
SAISIR prenom
AFFICHER "Bonjour "
AFFICHER prenom

organigramme



python

```
print("Saisis ton prénom")
prenom = input()
print("Bonjour ", prenom)
```

Nota : dans les organigrammes, les entrée/sortie sont représentées dans des parallélogrammes.

Rq1 : `prenom = input("Saisis ton prénom")` correspond à **AFFICHER** "Saisis ton prénom".
SAISIR prenom

Rq2 : Depuis `python3.6`, afin d'afficher une chaîne de caractères contenant des variables, on préfèrera utiliser la méthode `f`.

Ainsi la double instruction `print("Bonjour ", prenom)` deviendra `print(f"Bonjour {prenom}")` .

⚠ la fonction `input()` renvoie toujours une chaîne de caractères. Si l'on veut un nombre, il faudra transformer la chaîne de caractères composée des chiffres en un nombre via `int()` ou `float()`.

expl7 : on souhaite calculer l'Indice de Masse Corporel (IMC=poids / taille^2) d'une personne mesurant 1,60m.

algorithme **organigramme**

integer : p
float : imc
AFFICHER "Saisis ton poids (kg)"
SAISIR p
imc **PREND LA VALEUR** $p/(1{,}6^2)$
AFFICHER "Ton IMC est de "
AFFICHER imc

...

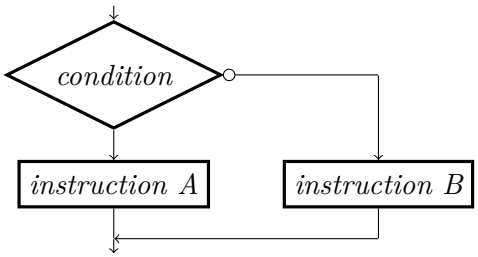
python

```
print("Saisis ton poids (kg)")
p = int(input())
imc = p / (1.6**2)
print("Ton IMC est de ", imc)
```

5 test conditionnel

algorithme **organigramme**

SI (la condition est vérifiée) **ALORS**
  | les instructions A sont effectuées
SINON
  | les instructions B sont effectuées
FIN-SI
Le reste du programme continue.



python

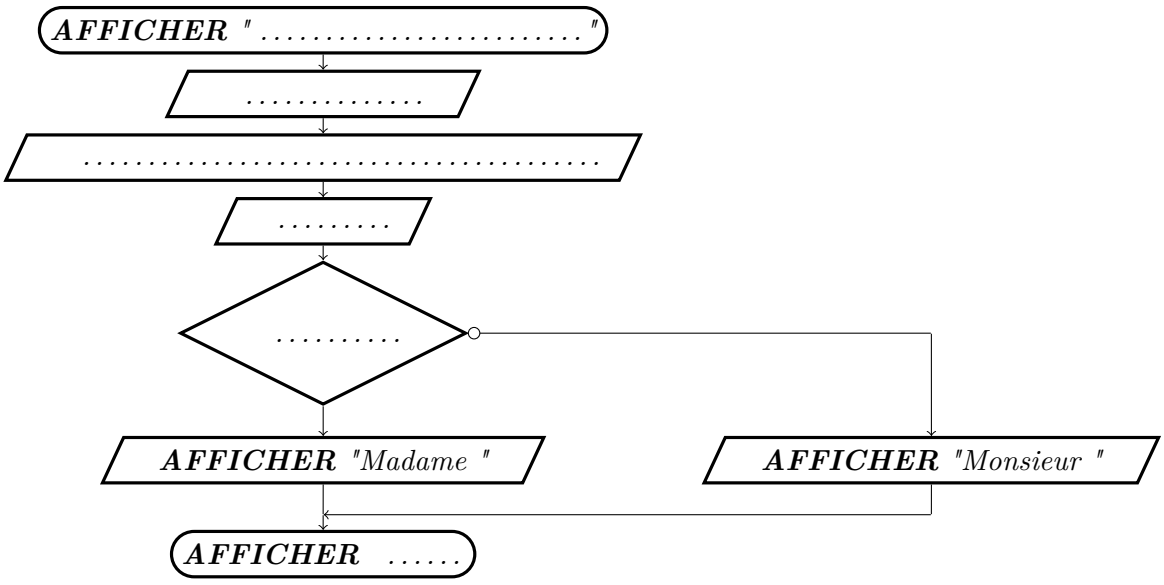
```
if (condition):            # test conditionnel
   bloc d instructions A   # est effectue si la condition est vraie
else :
   bloc d instructions B   # est effectue si la condition est fausse
instructions suivantes     # instructions a l'exterieur du bloc conditionnel
```

Nota : dans les organigrammes, les tests conditionnels sont représentées dans des losanges. expl8 : on souhaite afficher un message d'accueil personnalisé en fonction du genre de la personne. Analysons le problème.

* ...
* ...
* ...
  - ...
  - ...
  - ...

On peut proposer de coder le féminin par 'F' et le masculin par 'M'. Ainsi nous aurions,

organigramme



algorithme

python

```
"""
entrée : entrées clavier
sortie : affichage d'un message personnalisé
Selon le genre, "Madame" ou "Monsieur" est
↪  affiché avant le nom
"""
nom = input("Entrez votre nom")
g = input("Entrez votre genre ('F'/'M')")
if g=='F' :
   print("Madame ")
else :
   print("Monsieur ")
print(nom)
```

On propose comme jeu de tests (nom, g) : ("Dupont", 'F') ; ("Dupond", 'M') ; ("Duchamps", 'A'). Dérouler le programme sur ces 3 tests. Un problème survient, quel est-il ?

ligne n°	nom	g	g=='F'	sortie
1				

ligne n°	nom	g	g=='F'	sortie
1				

ligne n°	nom	g	g=='F'	sortie
1				

Que fait ce code ?

```
nom = input("Entrez votre nom")
g = input("Entrez votre genre ('F'/'M')")
if g=='F' :
   print("Madame ")
elif g=='M' :
   print("Monsieur ")
else :
   print("Genre inconnu pour ")
print(nom)
```

Quel code écrire afin de ne pas affchier le nom si l'utilisateur n'a entré ni 'F', ni 'H' ?

Définition 4

Un booléen est une entité qui ne peut prendre que ..... valeurs possibles, souvent .....

Récapitulatif des tests :

égalité	$\neq$	$>$	$\leqslant$	$\geqslant$	**inclus dans**	**ET**	**OU**
$x == 3$	$x != 3$	$x > 3$	$x >= 3$	$x <= 3$	x in "abc"	True and False	True or False

⚠ en python, - ne pas confondre l'affectation $x = 3$ avec le test $x == 3$,
- seul True représente Vrai (true, TRUE ne signifient rien).

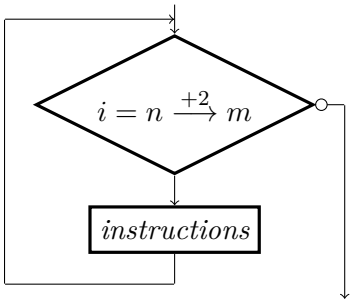
6 boucle finie

Afin de répéter une instruction un nombre déterminé de fois, nous utilisons une boucle POUR.

algorithme

POUR i **ALLANT DE** n **À** m **(pas** $+2$**) FAIRE**
  les instructions sont effectuées
FIN-POUR
Le reste du programme continue.

organigramme



python

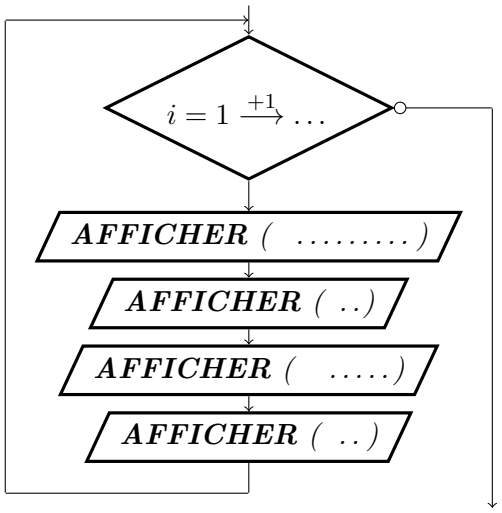
```
for i in range (n,m,2): # i prendra les valeurs entieres de 2 en 2 entre n et m-1.
   bloc d instructions   # bloc repete n-m fois
instructions suivantes   # instructions a l'exterieur de la boucle
```

Par défaut la valeur initiale est 0 et le pas est 1, ainsi `range(n)` correspond à `range(0,n,1)`.

⚠ `range(n)` correspond à l'ensemble {0 ;1 ;2 ; ... n-1}. La valeur finale n est exclue.

expl9 : on souhaite écrire la table de multiplication de 7 multiplié de 1 à 12.

organigramme **python**

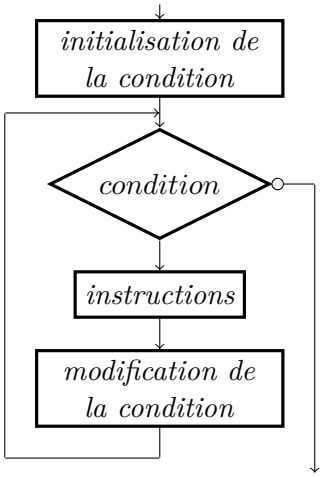


...

7 boucle conditionnelle

Lorsque le nombre de répétition est inconnu au départ ou peut-être modifié en cours d'éxécution, on utilise une boucle TANT-QUE.

initialisation de la condition
TANT-QUE (*condition*) **FAIRE**
    les instructions sont effectuées
    modification de la condition
FIN-TANT-QUE
Le reste du programme continue.



python

```
initialisation de la condition
while (condition) :
    bloc d instructions
    modification de la condition
instructions suivantes  # instructions a l'exterieur de la boucle
```

expl10 : on demande à l'utilisateur de résoudre une équation, tant qu'il n'a pas bien répondu on lui demande la réponse.

En python, et l'**organigramme**

...

```
print(f"Calculer 3*2 : ")
rep = int(input())
while rep!=6 :
    print(f"incorrect, 3*2= ?")
    rep = int(input())
printf("bravo !")
```

Comment modifier afin d'avoir le nombre d'essai avant d'obtenir la bonne réponse ?

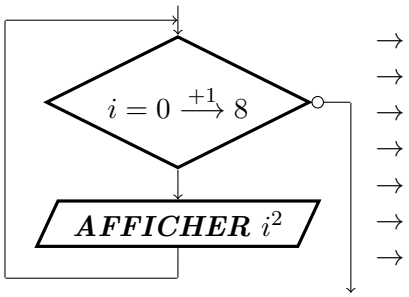
Théorème 1

Toute boucle POUR peut se traduire à l'aide d'une boucle TANT-QUE.

Démonstration : il suffit de définir le même compteur de boucle que la boucle POUR et faire le test de sortie dessus. □

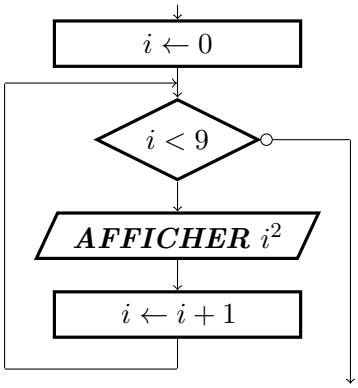
expl11 :

```
for i in range(9):
    print(i**2)
```



devient :

```
i=0
while i<9:
    print(i**2)
    i=i+1
```



8 tableau

Définition 5

Un tableau est ...

La taille du tableau est ...

expl12 :

0	1	2
'o'	'u'	'i'

Ce tableau contient des ..... et est de taille .....

0	1	2	3
12	17	14	16

Ce tableau contient des ..... et est de taille .....

Rq : en `python`, les notions de tableau et de liste chaînées sont similaires et on ne fera pas la différence. Ainsi ce genre de tableau est de type ..... Néanmoins, il existe des tableaux ..... comme les chaînes de caractères.

expl13 :

Ci-contre, deux tableaux ont été définis.
Le tableau `tab` est composé de ....., est de type .... et de longueur .....
tandis que la chaîne de caractères `chaine` est de type .... et de longueur .....
Pour obtenir la taille d'un tableau, on utilise la fonction `len()`.

```
tab = 3*[0]
chaine = "un texte"
type(tab)    # renvoye ...
len(tab)     # renvoye ...
type(chaine)# renvoye ...
len(chaine)  # renvoye ...
```

i Accès aux éléments sous `python`

On accède aux éléments du tableau à l'aide de crochets.
En `python`, le premier élément est indicé par .... Ainsi le dernier élément est indicé par la .....

expl14 :

```
tab[0] # renvoye ...
tab[0]=22
chaine[1] # renvoye ...
chaine[0]='U' # impossible, lève une erreur
```

Pour parcourir les éléments d'un tableau, il existe deux manières.

avec les **indices**

```
tab = [11,21,31]
for i in range(len(tab)):
    print(tab[i])
```

comme **itérateur**

```
tab = [11,21,31]
for e in tab:
    print(e)
```

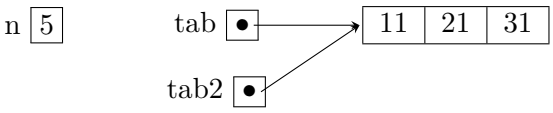
expl15 : écrire le code `python` pour épeler le mot `"Bonjour"`.

...

ii Copie et passage en argument

Rq : pour voir un fonctionnement imagé, on pourra se référer à l'outil en ligne [http://pythontutor.com/].

⚠ sous `python`, une variable référençant un tableau possède l'adresse mémoire du tableau. Contrairement aux variables numériques, aux booléens ou aux chaînes de caractères, où c'est la valeur qui est référencé.



◇ Conséquence n° 1 : écrire `tab=tab2` n'effectue pas une copie. Il faut utiliser la méthode `copy()`.

fausse copie

```
tab = [11,21,31]
tab2= tab
tab2[1]=0
print(tab) # affiche ...
```

vraie copie

```
tab = [11,21,31]
tab2= tab.copy()
tab2[1]=0
print(tab) # affiche ...
```

...

◇ Conséquence n° 2 :

Lorsqu'un tableau est passé en paramètre d'une fonction, si le code de la fonction modifie le tableau, le tableau initial est modifié.

expl16 :

```
def carree(tab, v) :
    for i in range(len(tab)):
        tab[i]=tab[i]**2
    v = v**2
```

```
v = 5
table = [1,2,3]
carree(table, v)
print(f"v={v}")          # affiche ...
print(f"table={table}") # affiche ...
```

...

9 fonctions

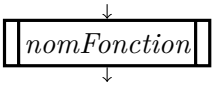
Dans un programme long ou complexe, afin de ..... du programme et donc les étapes de résolution, nous pouvons être amenés à découper le problème en plusieurs sous-problèmes. Chacun des sous-problèmes pourra être alors gérer par une procédure (/un programme) qui lui sera propre.
Si le programme est long, il est possible qu'un jeu d'opérations doive être effectué de nombreuses fois. On pourra alors vouloir ..... qui traite ce jeu d'opérations afin de ne pas devoir recopier les mêmes lignes.

Cela permet d'avoir un code plus ....., plus ..... et plus .....

sous `python`

```
def nomFonction(variables : type) -> type :
    bloc d'instructions
    return valeur
```

dans un organigramme



Rq : une fonction renvoie toujours une valeur, sinon il s'agit d'une procédure. Une procédure a des effets de bords (affichage, changement d'une variable, ...) mais ne renvoie pas de valeur (ou en `python` la valeur `None`).

Rq2 : définir le type des variable et le type de return est facultatif mais facilite la compréhension.

expl17 : si on souhaite calculer l'image de 3 par la fonction f définie par $f(x) = 3x^2 + 1$. On pourra définir la fonction f ainsi :

```
def f(x : int) -> int:
    return 3*x**2+1
```

Portée des variables

⚠ Toute variable définie dans une fonction est locale à la fonction. Sous `python`, si une variable est inconnue au niveau de la fonction, `python` va voir 'au-dessus' (à l'extérieur de la fonction) si elle existe.

Ainsi une fonction ne peut pas changer la valeur d'une variable a l'extérieur d'elle-même, à moins d'utiliser le tokken `global`. D'une manière générale, on essayera de proscrire l'utilisation de ce tokken.

expl18 :

```
def f(x : int) -> int:
    return 3*x+y

x=5
y=1
print(f(3))
```

affiche ...
car ...

expl19 :

```
def f(x : int) -> int:
    y=5
    return 3*x+y

x=5
y=1
print(f(3))
print(y)
```

affiche ...
car ...

expl20 :

```
def f(x : int) -> int:
    global y
    y=5
    return 3*x+y

x=5
y=1
print(f(3))
print(y)
```

affiche ...
car ...

Rq : on peut donner plusieurs paramètres à une fonction et celle si peut renvoyer plusieurs valeurs.

expl21 :

```
def mystere(a : float,b : float) -> tuple:
    if a<b :
        return a, b
    else :
        return b, a
```

Renvoie les nombres a et b de manière .....