Sous-programmes

Sous-programmes

Problème de lessive

Najla, Douja et Zohra ont fait leurs lessives aujourd'hui. Or, **Najla** fait sa lessive tous les 3 jours, **Douja** tous les 4 jours et **Zohra** tous les 6 jours.

Figure 1, Lessive

Questions

1. Combien passera-t-il de temps avant que les trois femmes ne refassent leurs lessives le même jour ?
2. En supposant que :
   - **Najla** fait la lessive tous les `a` jours. Avec `a > 0`
   - **Douja** fait la lessive tous les `b` jours. Avec `b > 0`
   - **Zohra** fait la lessive tous les `c` jours. Avec `c > 0`

   Déterminer quand les trois femmes referons leurs lessives le même jour ?
3. Ecrire l'algorithme d'un programme pour résoudre ce problème.

Solution

1. On pourra déterminer graphiquement le temps requis pour voir les trois femmes faire leurs lessives le même jour. Et ce en utilisant l'échelle temporelle suivante :

   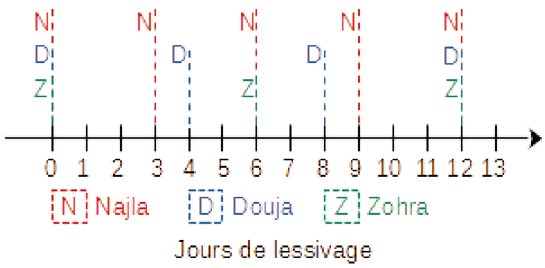

   Figure 2, Jours de lessive

   On en déduit qu'il faudra attendre **12 jours**.
2. On remarque que le temps requis pour voir les trois femmes faire leurs lessives dans une même journée peut être calculé en utilisant la formule suivante :

   ppcm(3, 4, 6) = ppcm(3, ppcm(4, 6)) = ppcm(ppcm(3, 4), 6)= 12

   Plus généralement le **temps requis pour voir les femmes faire leurs lessives la même journée** est :

   temps = ppcm(a, b, c) = ppcm(ppcm(a, b), c) = ppcm(a, ppcm(b, c))

   PPCM = Plus Petit Commun Multiple, c'est le plus petit nombre qui est multiple des trois nombres **a**, **b** et **c**.
3. Pour résoudre cet exercice il nous faudra calculer le **ppcm** de deux nombres. Une manière de faire est donnée dans l'algorithme suivant :

   Algorithme

```
ppcm ← a
TantQue ppcm mod b ≠ 0 Faire
  ppcm ← ppcm + a
Fin TantQue
```

L'algorithme du programme peut être écrit comme suit :

Algorithme

```
Algorithme lessive
Début
  Répéter
    Ecrire("Temps de lessive 1 ? ") ; Lire(a)
  Jusqu'à a > 0
  Répéter
    Ecrire("Temps de lessive 2 ? ") ; Lire(b)
  Jusqu'à b > 0
  Répéter
    Ecrire("Temps de lessive 3 ? ") ; Lire(c)
  Jusqu'à c > 0

  temps ← a
  TantQue temps mod b ≠ 0 Faire
    temps ← temps + a
  Fin TantQue
  TantQue temps mod c ≠ 0 Faire
    temps ← temps + a
  Fin TantQue

  Ecrire("Les femmes referont leurs lessives dans", temps, "jours")
Fin
```

Objet	Type/Nature
a, b, c, temps	entier

Amélioration de la solution

On remarque que la solution précédente **n'est pas très claire**, **trop longue**, et qu'elle **contient des duplications** (des répétitions).
Cette solution peut-être rendue **plus courte**, **plus claire** et **plus lisible** en **affectant des *noms* à certains *blocs d'instructions*** du premier algorithme.
Voici la nouvelle version de l'algorithme qui bénéficie de la **décomposition modulaire**.

Algorithme

```
Algorithme lessive_2
Début
  saisie(a)
  saisie(b)
  saisie(c)
  temps ← ppcm(ppcm(a, b), c)
  Ecrire("Les femmes referont leurs lessives dans", temps, "jours")
Fin
```

Objet	Type/Nature
a, b, c, temps	entier
saisie	procédure
ppcm	fonction

Décomposition modulaire

L'**analyse modulaire**, appelée également **décomposition modulaire**, consiste à diviser un problème en sous problèmes de difficultés moindres.
En algorithmique, les **sous problèmes** correspondent à des **sous-programmes**.

Sous-programme

Un **sous-programme** est une section de code nommée ***qui peut être appelée*** en écrivant le nom du sous-programme dans une instruction du programme.
Les **sous-programmes** sont également appelés **procédures** ou **fonctions**.
Une **procédure** exécute simplement un ensemble d'instructions, tandis qu'une **fonction** renvoie une valeur une fois son exécution est terminée.

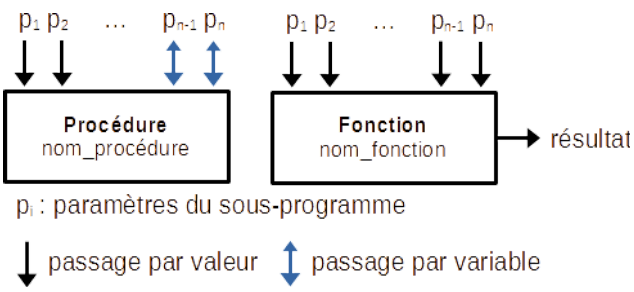

Figure 3, Différence entre une procédure et une fonction

L'écriture de **sous-programmes** rend le code plus lisible et réutilisable, car le code est subdivisé en des sections plus petites.
La plupart des langages de programmation sont livrés avec un ensemble de sous-programmes intégrés (fonctions prédéfinies). Ils permettent, aussi, au programmeur d'écrire leurs propres sous-programmes personnalisés.

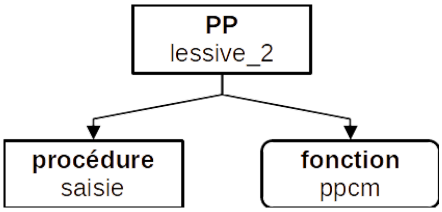

Figure 4, Décomposition modulaire du problème de lessive des trois femmes

Fonction

Définition

Une **fonction** est un sous-programme **qui retourne à son appelant un seul résultat** *en fonction de ses paramètres*.
Une **fonction** peut avoir zéro ou plusieurs paramètres. Ces **paramètres** sont ***souvent transmis par valeur***.

Appel

Comme une **fonction renvoie toujours une valeur**, son appel peut se faire de différentes manières :

- Dans une affectation :

Algorithme

```
// pgcd(a, b) renvoie le PGCD des deux valeurs
dc ← pgcd(a, b)
```

- Dans une structure conditionnelle :

Algorithme

```
// Afficher si un nombre est premier
Si premier(n) Alors
  Ecrire(n, "est premier")
Sinon
  Ecrire(n, "n'est pas premier")
Fin Si
```

- Dans une structure itérative :

Algorithme

```
// f(x) est une fonction qui admet un extrémum
// en x0 ∈ [0, +∞[
// Recherche de l'extrémum de f(x)
x0 ← 0
TantQue (f(x0+pas) > f(x0)) Faire
  x0 ← x0 + pas
Fin TantQue
```

Algorithme

```
// Saisir une chaine alphabétique
// est_alphabetique(ch) : retrourne Vrai
// Si ch[i] ∈ ["A", "Z"] U ["a", "z"]
Répéter
  Ecrire("Une chaine alphabétique ? ")
  Lire(ch)
Jusqu'à est_alphabetique(ch)
```

Algorithme

```
// Saisie d'un tableau de valeurs distinctes entre les indices n1 et n2
// Les fonctions :
// - min(a, b) : renvoie le minimum entre a et b
// - max(a, b) : renvoie le maximum entre a et b
// - existe(v, t, n) : recherche l'existence de v dans les n premières cases de t
Pour i de min(n1, n2) à max(n1, n2)-1 Faire
  Répéter
    Ecrire("t[", i, "] ? ")
    Lire(t[i])
  Jusqu'à (non existe(t[i], t, i-1))
Fin TantQue
```

- Comme paramètre d'un autre sous-programme :

Algorithme

```
// somme_carre(a, b) renvoie a² + b²
Ecrire(somme_carre(a, b))
```

Algorithme

```
// calculer PGCD de a, b et c
dc ← pgcd(a, pgcd(b, c))
```

Vocabulaire et Syntaxe

Une fonction s'écrit comme suit en algorithmique :

Algorithme

```
Fonction NomFonction(p1: type1, p2: type2, ...,
                     pn: typen):typerésultat
  //
  // Traitements
  //
  Retourner résultat
Fin
```

Son équivalent en Python s'écrit comme suit :

Python

```
def NomFonction(p1, p2, ..., pn):

    #
    # Traitements
    #
    return résultat
```

Procédure

Définition

Une **procédure** est un sous-programme **qui ne retourne pas, directement, de résultats <u>à son appelant</u>**.
Une **procédure** peut avoir zéro ou plusieurs paramètres. Ces **paramètres** peuvent être, selon le besoin, ***transmis par <u>valeur</u>*** ou ***transmis par <u>variable</u>***.
En algorithmique, le **mode de passage par variable** *est utilisé pour renvoyer, indirectement, un ou plusieurs résultats à l'appelant*.
Lorsqu'une procédure renvoie des résultats à travers ses paramètres, on dit qu'**elle possède un effet de bord**.

Appel

Comme une **procédure ne renvoie aucune valeur**, son appel se fait toujours de la même façon :

Algorithme

```
// Saisir une valeur dans n
saisir(n)
// Remplir le tableau t par n valeurs distinctes
remplir_tab(t, n)
// Echanger le contenu de deux variables
permuter(a, b)
```

Une **procédure** utilise les paramètres passés par valeur pour réaliser ses traitements. Elle peut, aussi, <u>modifier la valeur des paramètres transmis par variable</u>, directement, chez l'appelant.

Vocabulaire & Syntaxe

Une fonction s'écrit comme suit en algorithmique :

Algorithme

```
Procédure NomProcédure(p1: type1, p2: type2, ..., pn: type
  //
  // Traitements
  //
Fin
```

En Python, il n'y a pas d'équivalent pour une procédure. On utilise pour celà une fonction :

Python

```
def NomProcedure(p1, ..., pn):
    #
    # Traitements
    #
```

Paramètres et Mode de passage

Un **sous-programme** défini réellement un **<u>ensemble de traitements effectués sur des données</u>**. Ces **données** doivent être <u>passées au sous-programme</u> dans ***ses paramètres***.

Types de paramètres

On distingue deux types de paramètres :

- **Les paramètres formels :** utilisés lors de <u>la définition</u> d'un sous-programme.
- **Les paramètres effectifs :** utilisés lors de l'utilisation (<u>l'appel</u>) d'un sous-programme.

Dans la figure ci-dessous, les paramètres **a**, **b** et **c** qui figurent dans la <u>définition du sous-programme</u> **somme** sont appelés des paramètres **<u>formels</u>**
Les paramètres **x**, **y** et **z** utilisés dans le programme principal qui appelle le sous-programme **somme** sont dits **<u>effectifs</u>** ou **<u>réels</u>**.

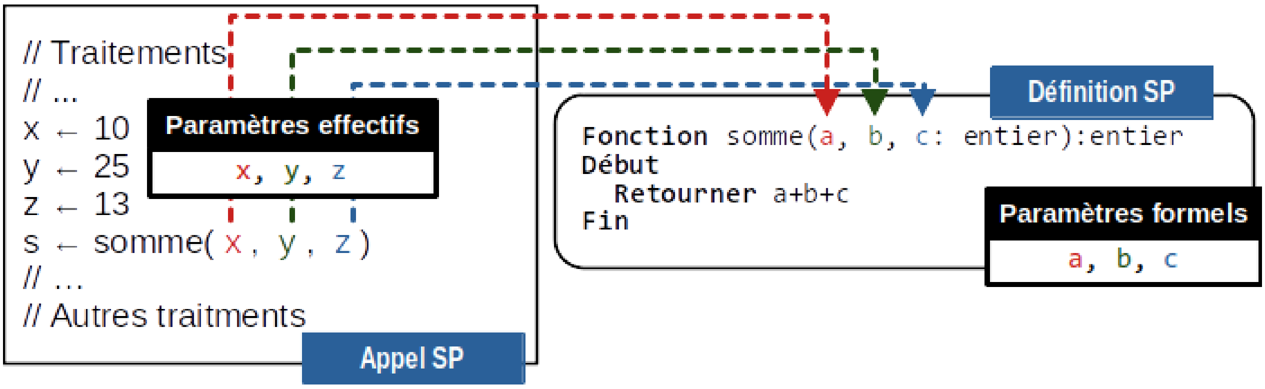

Figure 5, Les types de paramètres

Modes de passage des paramètres

Il existent deux modes de passage des paramètres : par variable ou par valeur.

Passage par valeur

Lors d'un **passage par valeur**, la valeur de l'expression passée en paramètre est copiée dans une variable locale. C'est cette variable qui est utilisée pour faire les traitements dans le sous-programme appelé.
Dans la figure suivante, la valeur de **y** est copiée dans le paramètre **x** de la fonction lors de l'appel de cette dernière.

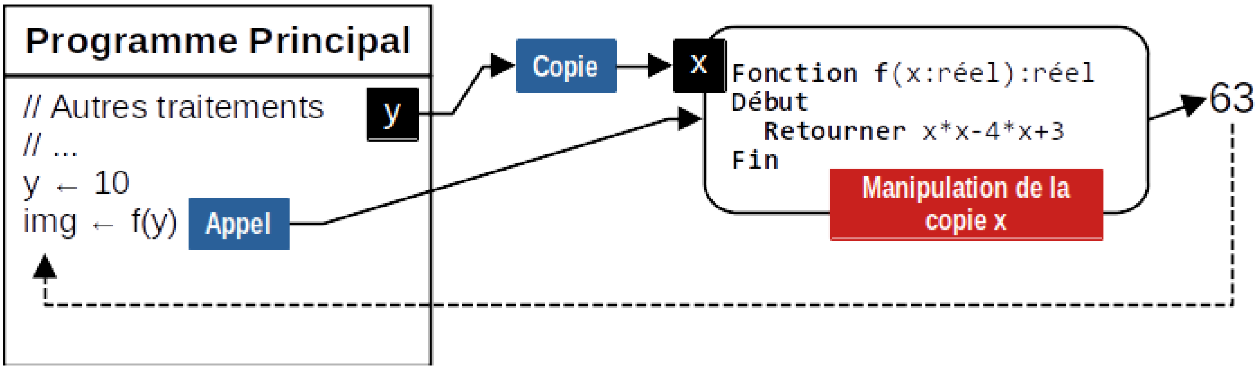

Figure 6, Passage par valeur

Passage par variable

Dans la figure suivante, l'adresse de la variable **y** est transmise à la procédure **saisie** sous le nom de **x**. Tout changement de la valeur de **x** dans la procédure à pour effet le changement de la valeur de **y**.
x et **y** sont deux noms différents de la même variable. La variable **y** dans le sous-programme est appelée **x**.

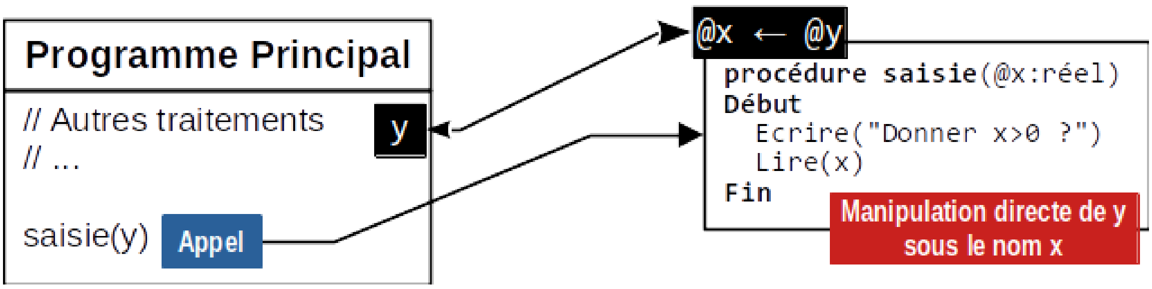

Figure 7, Passage par variable

Portée des variables

Toute **variable** a une durée de vie bornée **au bloc** où elle est déclarée. Ce bloc définit **la portée** de la variable.
Les **variables globales** sont déclarées dans le programme principal. Elles sont utilisables dans tout le programme.
Les **variables locales** sont déclarées dans un sous-programme. Elles ont une portée limitée et elles ne sont utilisables qu'à l'intérieur de celui-ci.
La fonction **existe** dans l'exemple suivant utilise les variables globales **t** et **n** qui sont déclarées dans le programme principal. **trouve** et **i** sont des variables locales car elles sont visibles uniquement à l'intérieur du sous-programme.

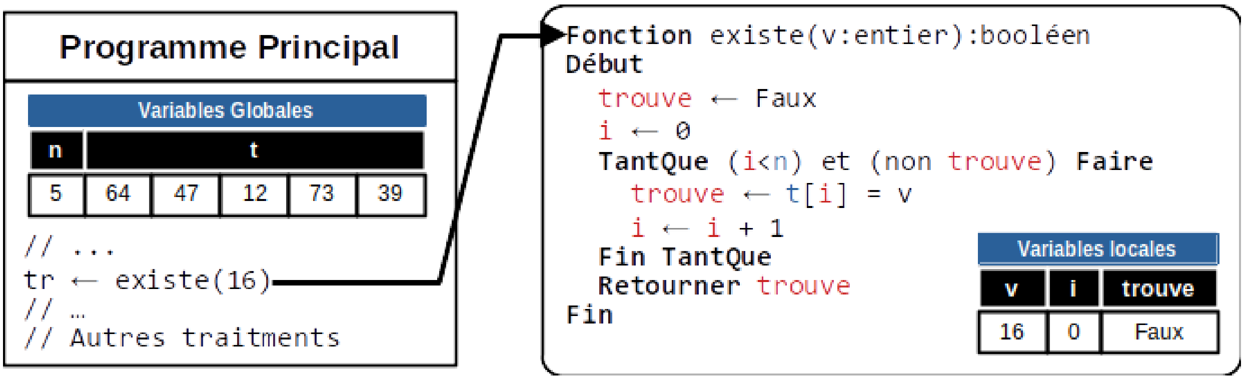

Figure 8, Variables globales et Variables locales

L'exemple précédent est **à éviter** car il contre-dit l'esprit de la modularité. **Il est conseillé vivement d'éviter les variables globales**.

Algorithme

```
// A éviter
// Dépend des variables du PP
Fonction existe(v:entier):booléen
  trouve ← Faux
  i ← 0
  TantQue (i<n) et (non trouve) Faire
    trouve ← t[i] = v
    i ← i + 1
  Fin TantQue
  Retourner trouve
Fin
```

Algorithme

```
// Meilleure écriture
// Indépendant du PP
Fonction existe(v:entier;t:tab;n:entier):booléen
  trouve ← Faux
  i ← 0
  TantQue (i<n) et (non trouve) Faire
    trouve ← t[i] = v
    i ← i + 1
  Fin TantQue
  Retourner trouve
Fin
```

Solution complète du problème de lessive

Problème

Najla, Douja et Zohra ont fait leurs lessives aujourd'hui. Or, **Najla** fait sa lessive tous les a jours, **Douja** tous les b jours et **Zohra** tous les c jours.

a, b et c sont des entiers strictement positifs.

Questions

1. Ecrire l'algorithme d'un programme pour résoudre ce problème. **Proposer une solution modulaire**.

Programme Principal

Le programme principal est la partie la plus importante d'un programme car c'est la partie qui sera exécutée lorsque l'algorithme sera traduit. Le code d'un sous-programme n'est exécuté que s'il est appelé dans le programme principal ou dans un autre sous-programme appelé par le programme principal.

Algorithme

```
Algorithme lessive_2
Début
  saisie(a)
  saisie(b)
  saisie(c)
  temps ← ppcm(ppcm(a, b), c)
  Ecrire("Les femmes referont leurs lessives dans", temps, "jours")
Fin
```

Objet	Type/Nature
a, b, c, temps	entier
saisie	procédure
ppcm	fonction

Procédure saisie

La procédure **saisie** permet de saisir un entier strictement positif.

Algorithme

```
procédure saisie(@n : entier)
Début
  Répéter
    Ecrire("Donner un entier > 0 ? ")
    Lire(n)
  Jusqu'à (n > 0)
Fin
```

Objet	Type/Nature
-	-

Fonction ppcm

La fonction **ppcm** retourne le **plus petit commun multiple** (ppcm) de deux entiers.

Algorithme

```
fonction ppcm(a, b : entier):entier
Début
  p ← a
  TantQue (p mod b ≠ 0) Faire
    p ← p + a
  Fin TantQue
  Retourner p
Fin
```

Objet	Type/Nature
p	entier

Programme Python

Python

```python
def saisie():
    n = 0
    while not (n > 0):
        n = int(input("Donner un entier > 0 ? "))
    return n

def ppcm(a, b):
    p = a
    while (p % b != 0):
        p += a
    return p

## Programme principal
a = saisie()
b = saisie()
c = saisie()
temps = ppcm(ppcm(a, b), c)
print("Les femmes referont leurs lessives dans", temps, "jours")
```

Application

Fraction irréductible

En mathématiques, une **fraction est irréductible** s'il n'existe pas de fraction égale ayant des termes plus petits. Autrement dit, une fraction irréductible **ne peut pas être simplifiée**.

Théorème

Soient **a** un entier et **b** un entier naturel non nul. Alors $\frac{a}{b}$ est irréductible si et seulement si **a** et **b** sont premiers entre eux.

Exemple

La fraction $\frac{12}{20}$ n'est pas irréductible car 12 et 20 sont des multiples de 4 : $\frac{12}{20} = \frac{3 \times 4}{5 \times 4} = \frac{3}{5}$ (simplification par 4). On peut aussi écrire $\frac{12}{20} = \frac{12 : 4}{20 : 4} = \frac{3}{5}$.

La fraction $\frac{3}{5}$ est irréductible car 1 est le seul entier positif qui divise à la fois 3 et 5.

Méthode de simplification

Pour réduire directement une fraction, il suffit de **diviser le numérateur et le dénominateur par leur plus grand commun diviseur**. D'après le lemme de Gauss, cette forme réduite est unique.

Exemple

Pour réduire la fraction $\frac{42}{390}$, on calcule $\mathrm{PGCD}(42, 390) = 6$ puis on simplifie par 6 : $\frac{42}{390} = \frac{6 \times 7}{6 \times 65} = \frac{7}{65}$.

Problème

On souhaite écrire l'algorithme d'un **programme modulaire** qui calcule la somme de deux fractions :

$$\frac{p1}{q1} + \frac{p2}{q2} = \frac{p1 \times q2 + p2 \times q1}{q1 \times q2} = \frac{p3}{q3} = \frac{\frac{p3}{pgcd(p3, q3)}}{\frac{q4}{pgcd(p3, q3)}} = \frac{ps}{qs}$$

*Figure 9, Somme de deux fractions avec : p1, p2, ps ∈ ℤ et q1, q2, qs ∈ ℤ**

Solution

Programme Principal

Le programme principal est la partie la plus importante d'un programme car elle fait appel aux différents sous-programmes qui ont été déclaré précédemment. Une **fonction** ou une **procédure** doivent être appelés pour résoudre un problème quelconque.

Algorithme

```
Algorithme Somme_Fraction
Début
  // Saisie des deux fractions
  saisie_fraction(p1, q1)
  saisie_fraction(p2, q2)
  // Simplifier les deux fractions
  simplifier_fraction(p1, q1)
  simplifier_fraction(p2, q2)
  // Calculer la somme des deux fractions
  // Puis la simplifier
  somme_fraction(p1, q1, p2, q2, ps, qs)
  // Afficher le résultat
  Ecrire(p1, "/", q1, "+", p2, "/", q2, "=", ps, "/", qs)
Fin
```

Objet	Type/Nature
p1, q1 p2, q2 ps, qs	entier
saisie_fraction simplifier_fraction somme_fraction	procédure

Procédure saisie_fraction

La procédure **saisie_fraction** permet à l'utilisateur d'introduire deux entiers qui représentent une fraction. Le dénominateur ne doit pas être null.

Algorithme

```
procédure saisie_fraction(@num, @denom: entier)
Début
  Ecrire("Numérateur ? "); Lire(num)
  Répéter
    Ecrire("Dénominateur ≠ 0 ? "); Lire(denom)
  Jusqu'à denom ≠ 0
Fin
```

Objet	Type/Nature
-	-

Procédure simplifier_fraction

La procédure **simplifier_fraction** utilise la méthode exposée au début de ce cours pour retrouver la fraction irreductible correspondant à la fraction saisie par l'utilisateur.

Algorithme

```
procédure simplifier_fraction(@num, @denom: entier)
Début
  dc ← pgcd(num, denom)
  num ← num div dc
  denom ← denom div dc
Fin
```

Objet	Type/Nature
dc	entier

Procédure somme_fraction

La procédure **somme_fraction** calcule la fraction irreductible correspondant à la somme de deux fractions.

Algorithme

```
procédure simplifier_fraction(p1, q1, p2, q2: entier;
                              @num, @denom: entier)
Début
  num ← p1 * q2 + p2 * q1
  denom ← q1 * q2
  simplifier_fraction(num, denom)
Fin
```

Objet	Type/Nature
-	-

Fonction pgcd

La méthode de simplification exposée dans ce cours nécessite le calcul du PGCD du numérateur et du dénominateur de la fraction à simplifier. C'est la fonction de **pgcd**.

Algorithme

```
Fonction pgcd(a, b: entier):entier
Début
  TantQue b ≠ 0 Faire
    r ← a mod b
    a ← b
    b ← r
  Fin TantQue
  retourner a
Fin
```

Objet	Type/Nature
r	entier

Programme Python

Python

```python
def saisie_fraction():
    p = int(input("Numérateur ? "))
    q = 0
    while not (q != 0):
        q = int(input("Dénominateur ≠ 0 ? "))
    return p, q

def pgcd(p, q):
    while q != 0:
        r = p % q
        p = q
        q = r
    return p

def simplifier_fraction(p, q):
    dc = pgcd(p, q)
    p = p // dc
    q = q // dc
    return p, q

def somme_fraction(p1, q1, p2, q2):
    p = p1 * q2 + p2 * q1
    q = q1 * q2
    return simplifier_fraction(p, q)

# PP
p1, q1 = saisie_fraction()
p2, q2 = saisie_fraction()

p1, q1 = simplifier_fraction(p1, q1)
p2, q2 = simplifier_fraction(p2, q2)
ps, qs = somme_fraction(p1, q1, p2, q2)

print(p1, "/", q1, "+", p2, "/", q2, "=", ps, "/", qs)
```