Les bases : exercices résolus en Python
Corrigé

Objectifs

- Raffiner des problèmes simples ;
- Écrire quelques algorithmes simples ;
- Savoir utiliser les types de base ;
- Savoir utiliser les instructions « élémentaires » : d'entrée/sortie, affichage, bloc...
- Manipuler les conditionnelles ;
- Manipuler les répétitions.

Exercice 1 : Conversion pouce/centimètre

Écrire un programme qui réalise la conversion pouce/centimètre d'une longueur saisie au clavier. Une longueur sera saisie comme un nombre réel suivi d'un caractère précisant l'unité. Les unités possibles sont le pouce (p), le centimètre (c) ou le mètre (m). Le programme affichera la longueur exprimée en pouce et en centimètre.

Voici des exemples d'exécution du programme :

```
Entrez une longueur : 1p
1 p = 2.54 cm

Entrez une longueur : 2m
78.7402 p = 200 cm

Entrez une longueur : 2km
0 p = 0 cm
```

Modifier le programme pour permettre la saisie de l'unité aussi bien en minuscules qu'en majuscules.

Solution :

```
R0 : Afficher une longueur saisie au clavier en pouces et en centimètres
```

Nous reprenons les jeux de test proposés dans le sujet. Nous ajoutons un jeu de test qui consiste à saisir une unité en centimètres. Par exemple, 5.08 cm qui doit donner 1 p = 5.08 p.

Lorsque l'on regarde les résultats d'exécution donnés dans le sujet (et qui servent donc de spécification pour le programme à écrire), on constate que, quelle que soit l'unité utilisée pour saisir la longueur, il faut afficher d'abord la longueur en pouces, puis son équivalent en centimètres. En conséquence, on en déduit qu'il faut calculer les deux longueurs (une dans chaque unité). Ainsi, après saisi de la longueur de l'utilisateur, nous allons la convertir dans les deux unités puis afficher la ligne d'équivalence.

```
1  R1 : Raffinage De « Afficher une longueur saisie au clavier... »
2    | Saisir la longueur          valeur: out Réel ; unité: out Caractère
3    | Calculer la longueur en pouces et en centimètres
4                                  valeur, unité : in
5                                  lg_p  : out Réel -- longueur en pouces
6                                  lg_cm : out Réel -- longueur en centimètres
7    | Afficher le résultat                lg_p, lg_cm : in
```

La seule étape délicate est la deuxième. Calculer les longueurs en pouces et en centimètres dépend de l'unité saisie par l'utilisateur. Il s'agit donc d'utiliser une conditionnelle et plus précisément de faire un traitement par cas. Puisque l'unité est un caractère, donc un type scalaire, nous pouvons utiliser un **Selon**.

Remarque : Pour traiter le cas des majuscules, il suffit d'ajouter le cas majuscule à côté du cas minuscule correspondant.

On aurait également pu convertir la longueur saisie en minuscule avant de calculer les longueurs en pouces et centimètres.

Remarque : On aurait également pu utiliser des **Si** et **SinonSi**. Cependant, la structure du programme est plus claire en utilisant un **Selon**. Ceci est d'autant plus vrai si l'on traite également les majuscules.

Voici le raffinage correspondant.

```
1  R2 : Raffinage De « Calculer la longueur en pouces et en centimètres »
2    | Selon unité Dans
3    |     'p', 'P':       { la longueur a été saisie en pouces }
4    |         lg_p <- valeur
5    |         lg_cm <- lg_p * UN_POUCE
6    |
7    |     'c', 'C':       { la longueur a été saisie en centimètres }
8    |         lg_cm <- valeur
9    |         lg_p <- lg_cm / UN_POUCE
10   |
11   |     'm', 'M':       { la longueur a été saisie en mètres }
12   |         lg_cm <- valeur * 100
13   |         lg_p <- lg_cm / UN_POUCE
14   |
15   | Sinon               { Unité non reconnue }
16   |     lg_p <- 0
17   |     lg_cm <- 0
18   | FinSelon
```

Notons que nous avons utilisé une constante symbolique (`UN_POUCE`) plustôt qu'une constante littéralle (2,54). L'intérêt est d'avoir un programme plus lisible et d'éviter la redondance (si on

se trompe sur la valeur d'un pouce en centimètres il suffit de faire un seule changement pour corriger le programme dans le cas de la constante symbolique contre trois sinon).

```
Algorithme pouce2cm

    -- Afficher une longueur saisie au clavier en pouces et en centimètres.

Constantes
    UN_POUCE = 2.54     -- valeur en centimètres d'un pouce
Variables
    valeur: Réel        -- valeur de la longueur lue au clavier
    unité: Caractère    -- unité de la longueur lue au clavier
    lg_cm: Réel         -- longueur exprimée en centimètres
    lg_p: Réel          -- longueur exprimée en pouces

Début
    -- saisir la longueur (valeur + unité)
    Écrire ("Entrer une longueur (valeur + unité) : ")
    Lire (valeur)               -- saisir la valeur
    Lire (unité)                -- saisir l'unité

    -- calculer la longueur en pouces et en centimètres
    Selon unité Dans
        'p', 'P':       { la longueur a été saisie en pouces }
            lg_p <- valeur
            lg_cm <- lg_p * UN_POUCE

        'c', 'C':       { la longueur a été saisie en centimètres }
            lg_cm <- valeur
            lg_p <- lg_cm / UN_POUCE

        'm', 'M':       { la longueur a été saisie en mètres }
            lg_cm <- valeur * 100
            lg_p <- lg_cm / UN_POUCE

    Sinon               { Unité non reconnue }
        lg_p <- 0
        lg_cm <- 0
    FinSelon

    -- afficher le résulat
    ÉcrireLn (lg_p, " p = ", lg_cm, " cm")
Fin
```

```
# -*- coding: utf-8 -*-
"""
ROLE : Afficher une longueur saisie au clavier en pouces et en centimètres.
EXEMPLE : 0.1 m => 3.937 pouces et 10 cm ; 10 pouces => 25.4 cm
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""

```

```
9  # CONSTANTE
10 UN_POUCE = float(2.54) # 1 pouce = 2.54 cm
11 # VARIABLES
12 valeur = float() # valeur de l longueur saisie au clavier
13 unite = str()    # valeur de l'unite saisie au clavier
14 longueurEnPouce = float() # longueur convertie en pouces
15 longueurEnCm = float()    # longueur convertie en centimètres
16 #
17 print ("Conversion d'une mesure en pouces ou en centimètres")
18 #1.Saisir la longueur (valeur + unité)
19 valeur = float(input("Saisir la valeur de la longueur : "))
20 unite = input("Saisir l'unité : p (pouce) , m (metre) , c (centimetre) ")
21 unite = unite.upper() # conversion en majuscule pour faciliter les tests
22 #2.Calculer la longueur en pouces et en centimètres
23 if unite == "P" :   # la longueur a été saisie en pouces
24     longueurEnPouce = valeur
25     longueurEnCm = valeur * UN_POUCE
26 elif unite == "M" : # la longueur a été saisie en mètres
27     longueurEnPouce = 100 * valeur / UN_POUCE
28     longueurEnCm = 100 * valeur
29 elif unite == "C" : # la longueur a été saisie en centimètres
30     longueurEnPouce = valeur / UN_POUCE
31     longueurEnCm = valeur
32 else : # unité non reconnue
33     longueurEnPouce = 0 ; longueurEnCm = 0
34 #3.Afficher le résultat
35 print ("Valeurs brutes :", longueurEnPouce, "pouces et", longueurEnCm, "cm")
36 print ("Valeurs arrondies :", \
37     format(longueurEnPouce, ".3f"), "pouces et", \
38     format(longueurEnCm, ".3f"), "cm")
```

Exercice 2 : Quelques statistiques

L'objectif de cet exercice est de calculer quelques statistiques sur une série de valeurs réelles. Cette série est lue au clavier. On considère qu'elle se termine par la valeur nulle (0) qui ne fait pas partie de la série.

2.1 Écrire un programme qui détermine la moyenne des valeurs de la série.

Solution : Pour calculer la moyenne, j'ai besoin de la somme des valeurs et du nombre de valeurs. On peut donc en déduire les deux variables d'accumulation suivantes :

```
Variables
    nb: Entier  -- le nombre de valeurs lues de la série
    somme: Réel -- la somme des valeurs lues de la série
```

Les premiers niveaux de raffinage peuvent être présentés ainsi :

```
R0 : Afficher la moyenne d'une série de valeurs réelles

R1 : Raffinage De « Afficher la moyenne d'une série de valeurs réelles »
      | Déterminer le nb d'éléments dans la série et leur somme
      | Afficher le résultat

R2 : Raffinage De « Déterminer le nb d'éléments dans la série et leur somme »
      | Initialiser : somme <- 0 et nb <- 0
      | Saisir le premier réel x
      | TantQue x <> 0 Faire
      |   | Mettre à jour les variables somme et nb en fonction de x
      |   | Saisir le réel suivant x
      | FinTQ

R2 : Raffinage De « Afficher le résultat »
      | Si nb > 0 Alors
      |   | Afficher la moyenne = somme / nb
      | Sinon
      |   | Signaler moyenne impossible
      | FinSi
```

Remarque : On pouvait pas utiliser un **Pour** car on ne connait pas à priori le nombre de valeurs dans la série. Ceci dépend de l'utilisateur. C'est lui qui décide de saisir 0. On doit donc utiliser soit un **TantQue**, soit un **Répéter**.

Nous avons utilisé un **TantQue** car nous avons considéré « traiter une valeur » qui peut ne pas être exécutée (cas où l'utilisateur saisit tout de suite 0 pour indiquer que la série est vide).

Nous aurions également pu utiliser un **Répéter** en nous intéressant à la saisie d'un réel. L'utilisateur doit nécessairement saisir au moins un réel (une valeur de la série ou zéro). Notons que dans ce cas, il faut ensuite faire un test pour savoir si le réel saisi fait partie de la série et, dans l'affirmative, le comptabiliser.

Voici le raffinage correspondant.

```
Initialiser les variables statistiques avec x
Répéter
  | Saisir un réel              x: out
  | Si x <> 0 Alors     { x est une valeur de la série }
  |   | Mettre à jour les variables statistiques
```

```
  | FinSi
Jusqu'à x = 0
```

On remarque que dans le cas du **TantQue** on duplique une instruction (la saisie d'un réel) et dans le cas du **Répéter** on duplique un condition.

L'algorithme peut ensuite être le suivant :

```
Algorithme statistiques_simples_moyenne

    -- Afficher la moyenne d'une série de valeurs réelles lues au clavier.
    -- La série est terminée par zéro qui n'appartient pas à la série.

Variables
    x: Réel     -- un réel lu au clavier
    nb: Entier  -- le nombre de valeurs dans la série
    somme: Réel -- la somme des valeurs lues de la série

Début
    -- Déterminer le nb d'éléments dans la série et leur somme
    -- Initialiser les variables
    somme <- 0  -- pas encore d'éléments lus
    nb <- 0
    -- Saisir le premier réel
    Lire(x)
    -- Traiter les éléments de la série
    TantQue x <> 0 Faire
        -- Mettre à jour les variables somme et nb en fonction de x
        somme <- somme + x
        nb <- nb + 1

        -- Saisir le réel suivant
        Lire(x)
    FinTQ

    -- Afficher le résultat
    Si nb > 0 Alors                    { La moyenne a un sens }
        -- Afficher la moyenne
        ÉcrireLn("Moyenne = ", somme / nb)
    Sinon                              { La moyenne n'a PAS de sens }
        -- Signaler moyenne impossible
        ÉcrireLn("La série est vide.  La moyenne n'existe donc pas !")
    FinSi
Fin.
```

```
# -*- coding: utf-8 -*-
"""
ROLE : Afficher la moyenne d'une serie de valeurs reelles lues au clavier
EXEMPLES :
    1  2  3  (0) --> 2.0
    2  -2  (0)   --> 0.0
    -4 -2  (0)   --> -3.0
    13  (0)      --> 13.0
```

```
    (0)          --> Non defini
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
x = float()          # chaque reel saisi au clavier
nb = int(0)          # le nombre de valeurs saisies de la serie, initialisé à 0
somme = float(0) # la somme des valeurs saisies de la serie, initialisée à 0
#
print ("Moyenne d'une série de valeurs")
#1.Afficher la consigne
print ("Donnez une serie d'entiers qui se termine par le nombre 0")
#2.Saisir le premier reel
x = float(input("Saisir un nombre (0 pour finir) : "))
#3.Traiter les éléments de la série
while x != 0 : # traiter le réel venant d'être saisi
    #3.1.Mettre à jour les variables somme et nb en fonction de x
    nb = nb + 1
    somme = somme + x
    #3.2.Saisir un nouveau réel x
    x = float(input("Saisir un nombre (0 pour finir) : "))
#4.Afficher le résultat
if nb > 0 : # la moyenne a un sens
    print ("Moyenne = ", somme/nb)
else : # la moyenne n'a PAS de sens
    print ("La série est vide. La moyenne n'existe donc pas !")
```

2.2 En plus de la moyenne, on veut connaître la plus petite et la plus grande valeur de la série.
Solution : Pour calculer la plus grande et la plus petite valeur, je pourrais procéder comme pour la moyenne en identifiant les deux variables min et max.

```
Variables
    min: Réel   -- la plus petite des valeurs lues de la série
    max: Réel   -- la plus grande des valeurs lues de la série
```

Le problème est de savoir avec quelle valeur intialiser max et min. Particulariser une valeur est toujours dangereux. Le mieux est donc de les initialiser avec la première valeur de la série... à condition que cette valeur existe. Le test de la série vide n'est donc plus fait *a posteriori mais a priori*.

Les premiers niveaux de raffinage sont alors :

```
R0 : Afficher des statistiques sur une série de valeurs réelles

R1 : Comment « Afficher des statistiques sur une série de valeurs réelles »
      | Saisir la première valeur x
      | Si x est nulle Alors
      |   | Indiquer que les statistiques demandées ne peuvent pas être faites
      | Sinon
      |   | Initialiser les variables statistiques avec x
      |   | Saisir une nouvelle valeur x
      |   | TantQue x est NON nulle Faire
      |   |   | Mettre à jour les variables statistiques
```

```
12        |   |   | Saisir une nouvelle valeur x
13        |   | FinTQ
14        |   | Afficher les statistiques
15        | FinSi
```

L'algorithme peut ensuite être le suivant :

```
1  Algorithme statistiques_simples_max
2
3      -- Afficher la moyenne, la plus grande et la plus petite valeur
4      -- d'une série de valeurs réelles lues au clavier.
5      -- La série est terminée par zéro qui n'appartient pas à la série.
6
7  Variables
8      x: Réel     -- un réel lu au clavier
9      nb: Entier  -- le nombre de valeurs lues de la série
10     somme: Réel -- la somme des valeurs lues de la série
11     min: Réel   -- la plus petite des valeurs lues de la série
12     max: Réel   -- la plus grande des valeurs lues de la série
13
14 Début
15     -- Saisir le premier réel x
16     Lire(x)
17
18     Si x = 0 Alors
19         -- Indiquer que les statistiques demandées ne peuvent être faites
20         ÉcrireLn("La série est vide.");
21         ÉcrireLn("Les statistiques demandées n'ont pas de sens !")
22     Sinon
23         -- Initialiser les variables statistiques avec x
24         max <- x
25         min <- x
26         somme <- x
27         nb <- 1
28
29         -- Saisir une nouvelle valeur x
30         Lire(x)
31
32         TantQue x <> 0 Faire
33             -- Mettre à jour les variables statistiques
34             nb <- nb + 1
35             somme <- somme + x
36             Si x > max Alors
37                 max <- x
38             SinonSi x < min Alors
39                 min <- x
40             FinSi
41
42             -- Saisir une nouvelle valeur x
43             Lire(x)
44         FinTQ
45
46         -- Afficher les statistiques
```

```
47          ÉcrireLn("Moyenne = ", somme / nb)
48          ÉcrireLn("Plus petite valeur = ", min)
49          ÉcrireLn("Plus grande valeur = ", max)
50      FinSi
51  Fin.
```

Exercice 3 : Division entière

Étant donnés deux entiers positifs, calculer le quotient et le reste de la division euclidienne du premier nombre par le deuxième. On utilisera uniquement l'addition et la soustraction sur les entiers.

Solution :

Remarque : En utilisant seulement l'addition et la soustraction, on ne peut pas utiliser une boucle **Pour**. Il faut donc choisir entre **TantQue** et **Répéter**.

```
Algorithme div_mod

    -- Calculer le quotient (div) et le reste (mod) de la division entière de
    -- deux entiers lus au clavier

Variables
    dividende: Entier   -- dividende lu au clavier, positif
    diviseur: Entier    -- diviseur lu au clavier, strictement positif
    reste: Entier       -- reste de la division entière
    quotient: Entier    -- quotient de la division entière

Début
    -- saisir le dividende et le diviseur avec contrôle
    ...

    -- calculer le quotient et le reste
    reste <- dividende
    quotient <- 0
    TantQue reste >= diviseur Faire
        { Variant : reste }
        { Invariant : diviseur * quotient + reste = dividende }
        quotient <- quotient + 1
        reste <- reste - diviseur
    FinTQ

    -- afficher le résultat
    ÉcrireLn(dividende, " / ", diviseur, " = ",
            quotient, " * ", diviseur, " + ", reste)
Fin.

--      5       2       -->     2 * 2 + 1
--      10      2       -->     5 * 2 + 0
--      5       0       -->     diviseur nul !
```

```
# -*- coding: utf-8 -*-
"""
ROLE : Afficher le quotient (div) et le reste (mod) de la division entière
    de deux entiers lus au clavier (sans utiliser les opérateurs // et %)
EXEMPLE :
    10  2       --> 5 * 2 + 0
    2   5       --> 0 * 5 + 2
    -2  5       --> Le dividende doit etre strictement positif
    5   -2      --> Le diviseur doit etre strictement positif
```

```
    5   0       --> Le diviseur doit etre strictement positif
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.0 - 04/2016
"""
# VARIABLES
dividende = int() # dividende saisi au clavier, positif
diviseur = int()  # diviseur saisi au clavier, strictement positif
reste = int()     # reste de la division entiere
quotient = int()  # quotient de la division entiere
#
print ("Quotient et reste d'une division entière")
#1.Saisir le dividende et le diviseur avec contrôle
saisieOK = False # car saisie non encore faite
while not saisieOK : # saisir dividende et diviseur et vérifier la validité
    #1a.Saisir le dividende et le diviseur
    dividende = int(input("Dividende : "))
    diviseur  = int(input("Diviseur : "))
    #1b.Verifier le dividende
    if (dividende < 0) :
        print ("Le dividende doit etre strictement positif.")
    #1c.Verifier le diviseur
    elif (diviseur <= 0) :
        print ("Le diviseur doit etre strictement positif.")
    else : # saisie correcte
        saisieOK = True
#2.calculer le quotient et le reste
reste = dividende ; quotient = 0 # initialisation
while reste >= diviseur : # incrémenter le quotient en mettant à jour le reste
    # Invariant : dividende = diviseur * quotient + reste
    quotient = quotient + 1
    reste = reste - diviseur
# Ici : (reste < diviseur) et (dividende = diviseur * quotient + reste)
#3.afficher le resultat
print (dividende, "/", diviseur, "=", quotient, "*", diviseur, "+", reste)
```

Exercice 4 : Nombres premiers

Écrire un programme qui permet à son utilisateur de saisir une valeur entière et qui, en retour lui indique si c'est un nombre premier ou pas.

Solution : Il s'agit d'afficher si un nombre saisi par l'utilisateur est premier ou non.

```
R0 : Afficher si un nombre saisi est premier ou non
```

Pour ce qui concerne les jeux de test, on peut vérifier si le programme fonctionne sur les premiers nombres, par exemple de 1 à 20 ou de 1 à 100.

Le premier niveau de raffinage est classique. Il permet de clairement séparer la partie calcul de la partie interaction avec l'utilisateur.

```
R1 : Raffinage De « Afficher si un nombre saisi est premier ou non »
  | Saisir un nombre                          n: out Entier
  | Déterminer si le nombre est premier       n: in ; premier: out Booléen
  | Afficher le résultat                      n, premier: in
```

Seule la deuxième étape mérite d'être détaillée. Un nombre premier est un nombre qui n'admet pas de diviseurs autres que 1 et lui-même. Nous allons en proposer plusieurs raffinements que nous allons comparer d'un point de vu performance (en nombre d'opérations arithmétiques).

Ainsi, étant donné un entier n, on peut regarder s'il est divisible par les entiers compris entre 2 et $n-1$. L'idée est d'essayer chaque entier. Si on trouve un diviseur le n n'est pas premier. Si on ne trouve pas de diviseur, n est premier. On se sert donc de la variable booléenne que l'on initialise à $VRAI$ et qui sera mise à faux dès que l'on trouve un diviseur. On pourrait donc l'écrire avec un **Pour**.

```
premier <- VRAI
Pour diviseur <- 1 JusquÀ diviseur = n-1 Faire
    Si diviseur est un diviseur de n Alors
        premier <- FAUX
    FinSi
FinPour
```

Ce qui s'écrit de manière équivalente :

```
premier <- VRAI
Pour diviseur <- 1 JusquÀ diviseur = n-1 Faire
    premier <- premier Et (diviseur est un diviseur de n)
FinPour
```

Si l'utilisation d'un **Pour** est possible, elle est cependant peu judicieuse. En effet, dès qu'on a trouvé un diviseur, on sait que le nombre n'est pas premier et il est inutile d'essayer d'autres diviseurs. Nous devons donc utiliser un **TantQue** ou un **Répéter**. Nous optons pour un **TantQue**.

Remarquons que nous avons en fait initialisé la variable `premier` avec `n <> 1` pour tenir compte du fait que 1 n'est pas un nombre premier.

Voici le raffinage correspondant.

```
R2 : Raffinage De « Déterminer si un nombre est premier »
  | premier <-  n > 1
  | diviseur <- 2
  | TantQue premier Et (diviseur < n - 1) Faire
  |     premier <- Non diviseur est un diviseur de n
```

```
6     |     diviseur <- diviseur + 1
7     | FinTQ
8
9   R3 : Raffinage De « diviseur est un diviseur de n »
10    | Résultat <- n Mod diviseur = 0
```

En fait, on sait qu'il n'est pas nécessaire de regarder les diviseurs au delà de $n/2$ car il ne peut pas y en avoir (trivial). On peut donc reformuler la condition du **TantQue**

```
| TantQue premier Et (diviseur <= n Div 2) Faire
```

Cette optimisation permet de réduire par 2 le nombre d'opérations. Cependant, on peut faire beaucoup mieux. En effet, si un nombre n'est pas premier il admet un diviseur et en fait au moins deux dont l'un au moins est inférieur ou égal à sa racine carrée.

```
| TantQue premier Et (diviseur <= racine carrée de n) Faire
```

Cette optimisation est bien meilleure que la précédente. En effet, si on considère un nombre premier supérieur à 10000, dans la version initiale, on doit considérer 10000 diviseurs, dans la première optimisation 5000 et seulement 100 dans la seconde.
Remarque : Pour éviter d'utiliser les réels et donc les problèmes d'arrondis (et éventuellement une perte de performance), on peut transformer l'expression

```
diviseur <= racine carrée de n
```

par (en élevant au carré)

```
diviseur * diviseur <= n
```

et, pour éviter les débordements (`diviseur * diviseur` peut dépasser la capacité des entiers), il est préférrable de réorganiser les calculs :

```
diviseur <= n Div diviseur
```

Une fois cette fransformation réalisée, on obtient donc :

```
| TantQue premier Et (diviseur <= n Div diviseur) Faire
```

On peut encore améliorer l'algorithme. En effet, si on sait que le nombre n'est pas divisible par 2, il est inutile d'essayer les diviseurs pairs (4, 6, 9, etc.). On peut se limiter aux diviseurs impairs (3, 5, 7, 9, 11...). Voici le raffinage correspondant.

```
1   R2 : Raffinage De « Déterminer si un nombre est premier »
2     | Si n = 2 Alors
3     |   premier <- VRAI
4     | Sinon
5     |   | premier <-  (n >= 2) Et Non (2 divise n)
6     |   | diviseur <- 3
7     |   | TantQue premier Et (diviseur <= n Div diviseur) Faire
8     |   |     premier <- Non diviseur est un diviseur de n
9     |   |     diviseur <- diviseur + 2
10    |   | FinTQ
11    | FinSi
```

Remarque : On pourrait se passer du **Si** en initialisant `premier` de la manière suivante :

```
premier = (n == 2) Ou ((n >= 3) Et (n Mod 2 <> 0))
```

Modifier le programme pour qu'il propose à l'utilisateur d'entrer une autre valeur à traiter ou d'arrêter le programme.

Solution : Le raffinage est le suivant :

```
R0 :  Indiquer si des entiers saisis au clavier sont premiers ou non

R1 :  Raffinage De « R0 »
      | Répéter
      |  | Saisir un entier
      |  | Indiquer le caractère premier de l'entier
      |  | Demander à l'utilisateur s'il veut continuer
      | JusquÀ réponse = 'n'
```

On peut en déduire le raffinage suivant :

```
Algorithme nb_premier

    -- Indiquer si un nombre est premier où non
    -- Possibilité de recommencer

Variables
    n: Entier           -- parcourir les entiers de 1 à max
    i: Entier           -- parcourir les diviseurs potentiels de n
    premier: Booléen    -- n est-il premier
    réponse: Caractère  -- réponse de l'utilisateur (o/n)

Début
    Répéter             -- analyser un nombre de l'utilisateur
        -- saisir le nombre
        ÉcrireLn ("J'indique si un nombre est premier")
        Écrire ("Le nombre : ")
        Lire (n)

        -- Déterminer si n est premier
        Si n <= 3 Alors
            premier <- n > 0            -- 0 n'est pas premier
        Sinon
            premier <- ((n mod 2) <> 0) -- n Non divisible par 2
                    Et  ((n mod 3) <> 0)        -- n Non divisible par 3
            i <- 3
            TantQue premier Et (i < n Div i) Faire
                    -- n peut encore être premier
                    -- et il reste des diviseurs potentiels
                i <- i + 2
                premier <- (n mod i) <> 0       --  n Non divisible par i
            FinTQ
        FinSi

        -- afficher le résultat
        Si premier Alors
            ÉcrireLn ("OUI, ", n, " est premier")
```

```
37          Sinon
38              ÉcrireLn ("NON, ", n, " n'est pas premier")
39          FinSi
40
41          -- Demander à l'utilisateur si il veut continuer
42          Écrire ("Encore un nombre (o/n) ? ")
43          Lire (réponse)
44      JusquÀ réponse = 'n'
45  Fin.
```

Exercice 5 : Nombres parfaits

Écrire un programme qui affiche tous les nombres parfaits compris entre 2 et 1000.

Un nombre parfait est un entier égal à la somme de ses diviseurs, lui exclu. Par exemple, 28 est un nombre parfait ($28 = 1 + 2 + 4 + 7 + 14$).

Solution :

```
R0 : Afficher les nombres parfaits compris entre 2 et Max, lu au clavier

R1 : Raffinage De « R0 »
      | Pour n <- 2 JusquÀ n = Max Faire
      |   | Si n est parfait Alors
      |   |   | Afficher n
      |   | FinSi
      | FinPour

R2 : Raffinage De « n est parfait »
      | Calculer la somme des diviseurs de n (autre que 1 et n)
      | Résultat <- n = somme des diviseurs

R3 : Raffinage De « Calculer la somme des diviseurs de n »
      | somme <- 1
      | Pour i <- 2 JusquÀ racine carrée de n Faire
      |   | Si i diviseur de n Alors
      |   |   | somme <- somme + i + (n Div i)
      |   | FinSi
      | FinPour
      | Si n est un carré parfait Alors
      |   | somme <- somme - racine carrée de n
      | FinSi
```

```
# -*- coding: utf-8 -*-
"""
ROLE : Afficher les nombres parfaits inférieurs à une borne saisie au clavier.
    Un nombre est parfait s'il vaut la somme de ses diviseurs, lui-même exclu.
    Si c'est le cas, on affichera aussi la liste des ses diviseurs.
EXEMPLES : nombres parfaits inférieurs à 1000 :
    6 est parfait : 6 = 1+2+3
    28 est parfait : 28 = 1+2+4+7+14
    496 est parfait : 496 = 1+2+4+8+16+31+62+124+248
AUTEUR : Claude Monteil <monteil@ensat.fr>
VERSION : 1.1 - 04/2016
"""
# VARIABLES
borneMax = int() # nombre maximal qui sera testé
nombre = int()   # nombre courant (entre 2 et la borne) à tester
sommeDiviseurs = int() # somme courante des diviseurs de nombre
listeDiviseurs = str() # liste courante des diviseurs de nombre
diviseur = int() # diviseur courant
#
print ("Affichage des nombres parfaits inférieurs à une borne")
#1.Saisir la borne maximale de recherche
```

```
22 borneMax = int(input("Borne␣maximale␣de␣recherche␣:␣"))
23 for nombre in range(2,borneMax+1) : # tester si nombre est parfait
24     #2.calculer la somme des diviseurs de nombre (lui-même exclu)
25     sommeDiviseurs = 1 ; listeDiviseurs = "1" # 1 divise tout nombre
26     for diviseur in range(2,nombre//2+1) : # tester si diviseur divise nombre
27         if nombre % diviseur == 0 : # si oui, cumuler ce diviseur
28             sommeDiviseurs = sommeDiviseurs + diviseur
29             listeDiviseurs = listeDiviseurs + "+" + str(diviseur)
30     if (nombre == sommeDiviseurs) :
31         print (nombre, "est␣parfait␣:", nombre, "=", listeDiviseurs)
```