# Chapitre 3

# Calculs répétitifs et devinette

## 3.1 Additionner plusieurs nombres

L'un des exercices du premier chapitre consistait à additionner quatre nombres.

```
/* == Additionner 4 nombres == */
Variables Entières iNombre1, iNombre2, iNombre3, iNombre4
Variable Entière iSomme

Début du traitement
   /* Lecture des données */
   Afficher "Entrez le premier nombre : "
   Lire iNombre1
   Afficher "Entrez le deuxième nombre : "
   Lire iNombre2
   Afficher "Entrez le troisième nombre : "
   Lire iNombre3
   Afficher "Entrez le quatrième nombre : "
   Lire iNombre4

   /* Calcul de la somme des 4 nombres */
   iSomme Devient iNombre1+iNombre2+iNombre3+iNombre4

   /* Affichage du résultat */
   Afficher "La somme vaut "
   Afficher iSomme
Fin du traitement
```

La solution, bien que correcte, présente plusieurs inconvénients. La lecture des nombres est fastidieuse à rédiger et, surtout, la solution manque de généralité. On voit clairement qu'on devrait pouvoir déterminer le nombre de termes au moment de l'exécution et ainsi pouvoir, à l'aide d'un même programme, calculer la somme de trois nombres ou de vingt-cinq.

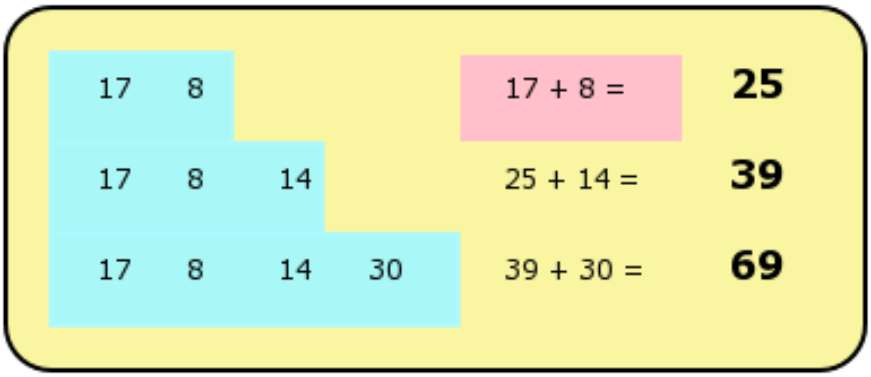
Pour parvenir à une telle solution, nous allons devoir augmenter notre stock d'opérations élémentaires et reconsidérer totalement notre solution. D'une manière évidente, nous allons remplacer notre traitement séquentiel lourd par un traitement simplifié, répété plusieurs fois.

## 3.2 Solution générale

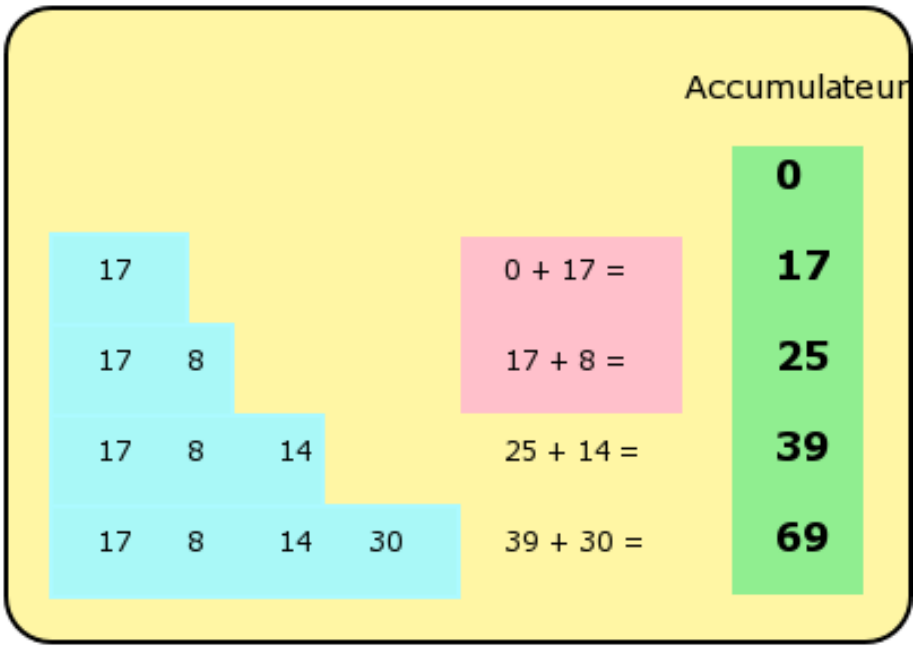
### 3.2.1 Variables et accumulateur

A partir du moment où le programme doit pouvoir additionner un nombre indéterminé de termes, il nous faut bien entendu une variable pour mémoriser ce nombre. Nous l'appellerons `iNombreTermes`. Par contre, il n'est plus possible de mémoriser chacun des termes, puisqu'il faudrait déclarer autant de variables `iNombreX` qu'il y aura de termes. La déclaration se faisant avant le début du traitement, nous pouvons pas savoir de combien de variables nous aurions besoin.

Une technique simple consiste à utiliser un boucle et un accumulateur. Un accumulateur est une variable qui prend une valeur initiale précise et se modifie plusieurs fois pour s'approcher petit à petit de la solution. Quand nous additionnons plusieurs nombres, c'est comme ça que nous procédons. Pour la série 17, 8, 14, 30, nous calculons successivement :




Les deux dernières opérations consistent à reprendre la somme précédente et à y ajouter le terme suivant. Par contre, la première opération est différente. Nous pouvons la remplacer par deux autres opérations :

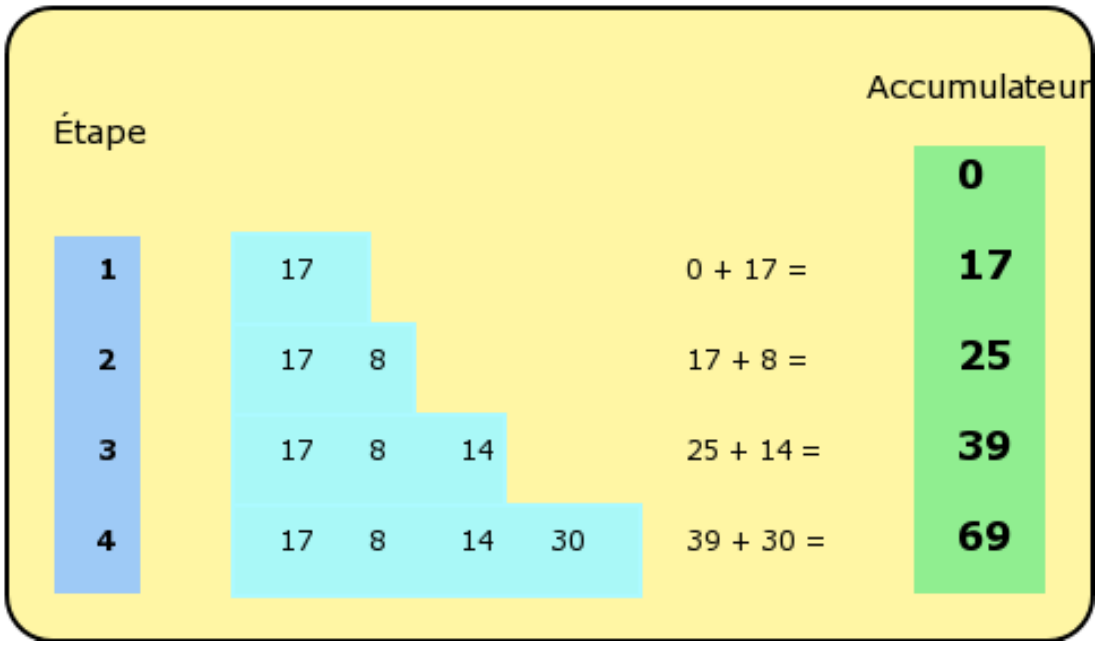



Il est désormais possible de définir chaque étape de la manière suivante :

*prendre le résultat de l'opération précédente et y ajouter le terme suivant.*

Il faut bien entendu que lors de la première opération, le résultat de l'*étape précédente* soit égal à 0. Ce résultat de l'opération précédente sera mémorisée dans un *accumulateur*. On dira qu'il faut initialiser l'accumulateur [1]. Nous nommerons cette variable `iSomme`.

Il reste un troisième point à préciser : lorsque nous procédons à une activité répétitive, il faut que nous gardions trace du nombre d'opérations effectuées. En effet, si nous devons demander cinq nombres à l'utilisateur, il faut que nous sachions à quel moment nous devons nous arrêter. Nous allons simplement compter les nombres à l'aide de la variable `iEtape`.




Enfin, il faudra évidemment lire chaque terme et pour cela disposer d'une variable pour le ranger (`iTerme`).

Nous sommes à présent en mesure de définir nos variables :

```
Variable Entière iNombreTermes
Variable Entière iSomme /* Accumulateur */
Variable Entière iEtape
Variable Entière iTerme
```

### 3.2.2 Traitement à effectuer

Notre programme se décompose clairement en trois parties :

```
Début du traitement
/* Définition du nombre d'opérations */
/* Lectures et calculs */
/* Affichage du résultat */
Fin du traitement
```

La première et la dernière partie sont triviales :

1. On peut dire que la valeur initiale d'un accumulateur est toujours l'élément neutre de l'opération qui le concerne. Ce sera donc 0 pour l'addition et 1 pour la multiplication.

```
Début du traitement

/* Définition du nombre d'opérations */'
  Afficher "Combien de nombres voulez-vous additionner ? "
  Lire iNombreTermes

/* Lectures et calculs */
/* ...À préciser ultérieurement... */

/* Affichage du résultat */
  Afficher "La somme des nombres vaut : "
  Afficher iSomme

Fin du traitement
```

Les lectures et les calculs vont être réalisés au moyen d'une boucle. Dans une boucle, nous devrons préciser plusieurs choses :

1. les valeurs initiales à placer dans certaines variables pour que la boucle se déroule dans de bonnes conditions.
2. la condition qui va nous permettre d'évaluer si la boucle doit continuer ou non.
3. le moment où il faudra évaluer si la boucle doit continuer ou non. L'importance de ce dernier point n'est pas toujours facile à percevoir.
4. les instructions qui doivent se répéter un certain nombre de fois.
5. faire quelque chose pour que la boucle finisse par se terminer.

### Valeurs initiales

Pour que notre solution soit correcte, il faut que l'accumulateur (`iSomme`) reçoive une valeur initiale précise. Nous avons vu qu'il devait recevoir la valeur 0. En outre, il faut que notre compteur d'étapes (`iEtape`) représente l'étape en cours. Nous lui donnerons la valeur initiale 1.

```
iSomme <- 0
iEtape <- 1
```

### Condition

Nous ne savons pas combien de fois nous allons parcourir la boucle, puisque cela dépend des intentions de l'utilisateur. Nous devrons donc comparer notre compteur d'étapes, avec le nombre d'opérations à réaliser. Tant qu'il sera inférieur ou égal au nombre d'opérations à réaliser, il nous faudra continuer.

```
iEtape <= iNombreTermes
```

### Moment d'évaluation de la condition

Il peut sembler sans importance de tester si le nombre d'opérations voulu ont été réalisées au moment d'entrer dans la boucle ou au moment d'en sortir. Il existe pourtant un cas particulier où c'est important : si l'utilisateur répond 0 (ce qui a du sens), il ne faut en fait effectuer aucune lecture ou aucun calcul. Il convient donc de mettre l'évaluation de la condition au début de la boucle.

**Instructions à répéter**

A chaque étape, nous allons demander le terme suivant et l'additionner à la somme déjà obtenue.

```
Afficher "Nombre suivant : "
Lire iTerme
iSomme <- iSomme + iTerme
/* une instruction manque */
```

**Moteur de progression de la boucle**

Nous n'avons oublié qu'un léger détail : préciser qu'à chaque étape, nous devons augmenter la valeur de la variable `iEtape`. En effet, si nous voulons que notre condition finisse par devenir fausse un jour, il faut que `iEtape` grandisse à chaque *itération*. Donc, en définitive, notre corps de boucle doit être le suivant

```
Afficher "Nombre suivant : "
Lire iTerme
iSomme <- iSomme + iTerme
Moteur Incrémenter iEtape
```

## 3.3 Structure d'une boucle et variantes

Les quatre premiers éléments de notre boucle vont correspondre à quatre parties d'une structure de boucle. Le moteur de progression est malheureusement plus difficile à formaliser. Il doit rester le cinquième élément auquel nous devrons penser à chaque fois. Une petite question devra chaque fois être posée : *qu'est-ce qui fait progresser la boucle vers sa fin ?*

### 3.3.1 Boucles à test initial

Voici ce que donnera notre boucle :

```
Initialisation
   iSomme <- 0
   iEtape <- 1
Tant que
   iEtape <= iNombreTermes
Répéter
   Afficher "Nombre suivant : "
   Lire iTerme
   iSomme <- iSomme + iTerme
   Moteur Incrémenter iEtape
Fin de boucle
```

Nous retrouvons trois zones qui contiendront des instructions ou des expressions :

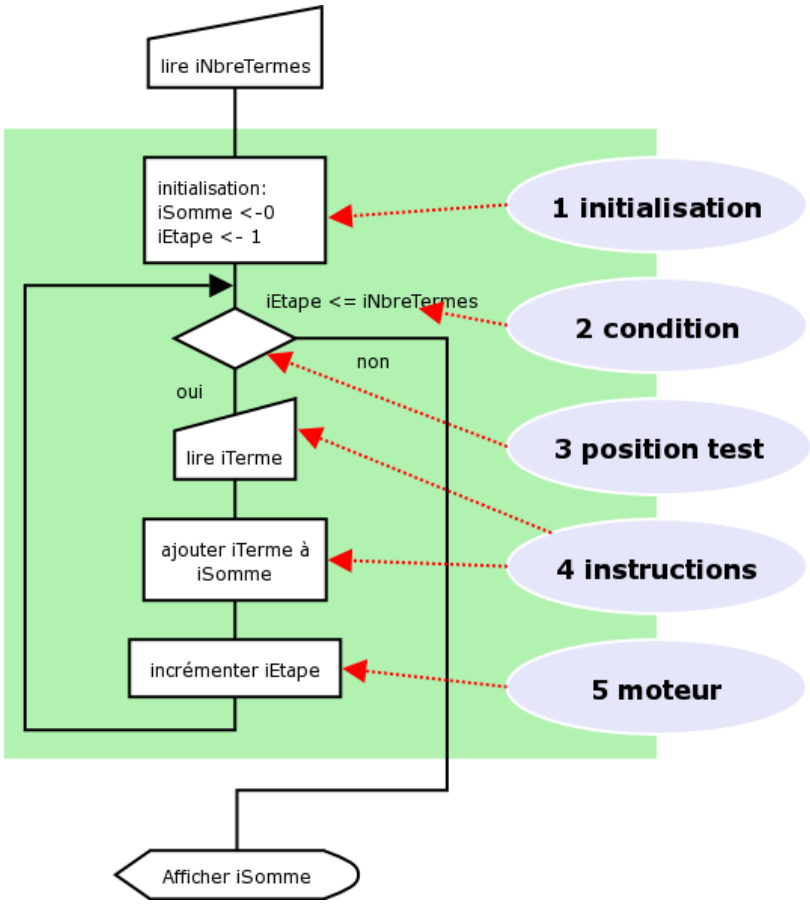



FIGURE 3.1 – Boucle à test initial

- la zone d'initialisation (commençant par `Initialisation` ou par `Init`) qui contient les initialisations de la boucle. Dans certains cas, rares, elle peut être vide. Ce n'est pas un problème. Les instructions d'initialisation sont toujours effectuées une seule fois.
- la zone reprenant l'expression de la condition initiale
- le corps de la boucle proprement dit qui doit contenir un ou plusieurs instructions qui finiront par rendre fausse la condition (j'ai ici souligné l'instruction qui incrémente `iEtape`). Je la fais précéder du mot `Moteur` pour signifier que c'est l'instruction qui fait progresser la boucle.

La position de la condition, avant le corps de la boucle, manifeste bien qu'elle sera évaluée avant d'exécuter la boucle. Au cas où la condition doit être évaluée à la fin d'une boucle, nous utiliserons une autre notation. Nous en verrons un exemple dans la sous-section suivante.

### 3.3.2 Boucles à test final

Dans beaucoup de pseudo-codes et dans certains langages, il n'existe pas de boucles à test final. Lorsque le test de boucle se fait à la fin de son exécution, on peut dire que le nombre d'itérations sera compris entre 1 et N, alors que, dans les boucles à test initial, le nombre d'itérations est compris entre 0 et N. Prenons comme exemple l'affichage des multiples de 3 en demandant après chaque nombre si l'utilisateur veut continuer.

```
1 /*== Affichage des multiples de 3 ==*/
2 Variable entière iMult3
```

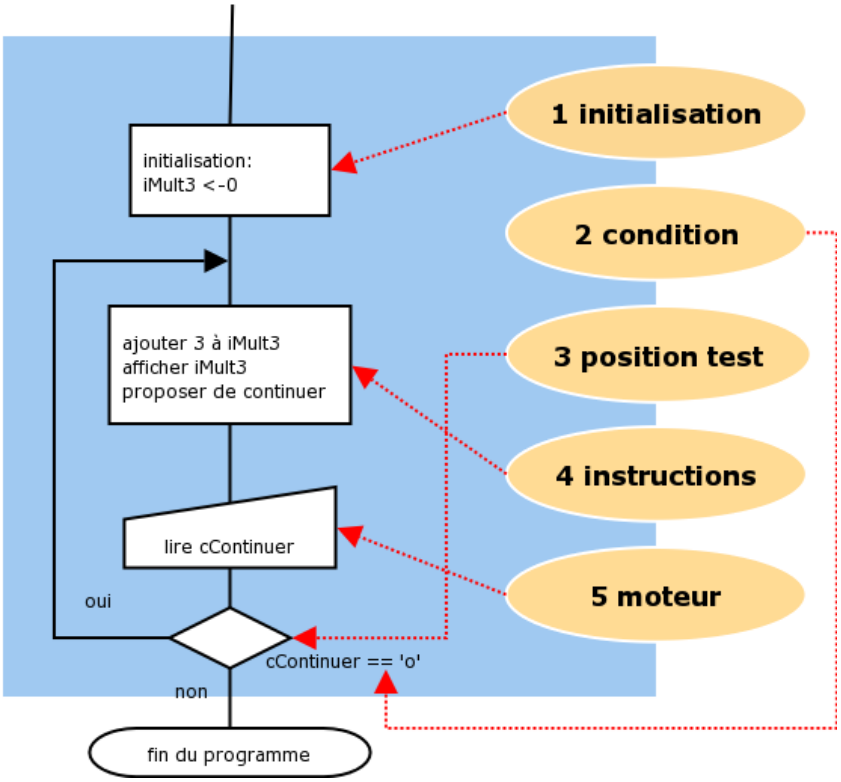



FIGURE 3.2 – Boucle à test final

```
3  Variable caractère cContinuer
4  Début du traitement
5     Initialisation
6        iMult3 <- 3
7        Afficher iMult3
8        Afficher "Voulez-vous continuer ?"
9        Lire cContinuer
10    Tant que cContinuer == 'o'
11    Répéter
12       iMult3 <- iMult3 + 3
13       Afficher iMult3
14       Afficher "Voulez-vous continuer ?"
15       Moteur Lire cContinuer
16    Fin de boucle
17 Fin du traitement
```

Dans ce programme, les instructions 7 à 9 sont en fait exprimées deux fois, ce qui constitue une perte de temps et une cause d'erreur si on décide plus tard de modifier le traitement. Nous allons exprimer que la condition est finale et faire l'économie de cette répétition.

```
Variable entière iMult3
Variable caractère cContinuer
Début du traitement
   Initialisation
      iMult3 <- 0
   Répéter
      iMult3 <- iMult3 + 3
      Afficher iMult3
```

```
        Afficher "Voulez-vous continuer ?"
        Moteur Lire cContinuer
    Boucler tant que cContinuer == 'o'
Fin du traitement
```

Le lecteur astucieux aura remarqué une différence importante supplémentaire entre les deux versions : comme l'incrémentation de `iMult3` se fait avant le premier affichage, il faut changer la valeur initiale de la variable. On aurait pu garder la valeur initiale et déplacer l'instruction d'incrémentation. Le passage d'un type de boucle à l'autre doit donc se faire avec prudence. La programmation ne consiste pas à appliquer aveuglément des recettes !

## 3.4 Jeu de devinette

Notre dernier exemple va consister à créer un jeu, simple mais connu. Il s'agit pour le joueur de deviner un nombre inventé par l'ordinateur en un nombre d'essais aussi petit que possible. Je laisserai au lecteur le soin d'imaginer une stratégie pour gagner, nous nous bornerons à écrire le programme permettant de jouer.

Précisions les contraintes :

- le nombre à deviner se trouvera dans une fourchette prédéfinie, entre 1 et 100 ;
- le joueur doit parvenir à trouver la solution en un maximum de 15 essais [2] ;
- à chaque réponse du joueur, l'ordinateur va donner une indication sur la qualité du nombre proposé : il peut être trop grand, trop petit ou constituer la solution ;
- le jeu doit se terminer quand le joueur a trouvé ou qu'il a épuisé ses possibilités d'essais.

### 3.4.1 Détermination des variables et définition de constantes

Notre programme va évidemment manipuler deux variables pour mémoriser le nombre mystérieux et les solutions successives proposées par l'utilisateur. Si nous voulons compter (et contrôler) le nombre d'essais, il nous faudra également une variable (une sorte de compteur).

Enfin, il apparaît que notre programme utilise deux nombres particuliers, qui jouent un rôle capital dans notre programme : la limite supérieure de l'intervalle de choix (100) et le nombre maximum d'essais. Nous allons nous simplifier la vie en leur donnant des noms. Comme ces valeurs ne changeront pas, nous allons les appeler des constantes. Les constantes permettent de faciliter la lecture du programme (un lecteur pressé peut se demander à quoi correspond le nombre 15 dans une comparaison). Elles permettent aussi une modification ultérieure plus facile. Si nos joueurs échouent tous en jouant avec le programme, nous pourrons choisir de l'adapter et de proposer 20 essais. Il suffit alors de modifier la déclaration de la constante, au début du programme, sans devoir relire tout le programme. Cette technique est particulièrement utile si une même constante apparaît plusieurs fois dans un programme. Par convention, les constantes s'écriront en majuscules.

**Constante** ***NOMCONSTANTE = Valeur***

```
Constante ESSAIMAX = 15
Constante NOMBREMAX = 100
Variable Entière iMystere
Variable Entière iProposition
```

2. On peut démontrer qu'un maximum de 8 essais est nécessaire pour y parvenir, à condition de trouver la bonne stratégie.

```
Variable Entière iNombreEssais
```

### 3.4.2 Préparation du jeu (génération du nombre aléatoire)

La génération d'un nombre aléatoire n'est pas une chose aisée. En général, il faut créer une fonction mathématique qui produit des séries de nombres non convergents (ce qui n'est pas simple). A titre d'exemple, la formule suivante possède cette propriété [3].

$$x_{n+1} = 23.x_n - int(\frac{23.x_n}{10^8+1}).(10^8+1)$$

La formule, pour un $x_0$ égal à 0, donne la série suivante :

```
10100381   32308761   43101496   91334399     691156
15896588   65621521    9294968   13784262   17038023
```

Il n'est heureusement plus nécessaire de programmer de telles formules. Elles sont préprogrammées dans les bibliothèques fournies avec les compilateurs. Le principe est simple : on possède une fonction (l'équivalent de la partie droite de la formule) qui a conservé trace du dernier nombre généré et l'utilise comme « semence » pour calculer le suivant. Malheureusement, les séries ainsi obtenues sont toujours identiques à chaque exécution du programme. Pour éviter cela, on peut réinitialiser le mécanisme au moyen d'une valeur différente : la date et l'heure sont idéales. La commande `Aléatoire` s'utilise une fois dans chaque programme et se sert de l'heure pour initialiser le générateur aléatoire. Quand à la fonction `HasardMax()`, elle renvoie un nombre aléatoire compris entre 0 et son argument. Pour générer un nombre compris entre 1 et 100, il faut utiliser `1+HasardMax(100)`.

```
Aléatoire
iMystere <- HasardMax(NOMBREMAX) + 1
```

### 3.4.3 Déroulement du jeu

#### Initialisation

La seule chose qu'il faille prévoir afin que notre jeu se déroule correctement est l'initialisation du compteur d'essai. Nous allons donner à ce compteur la signification de nombre d'essais réalisés. Au départ, il vaut donc 0. Quand nous sortirons de la boucle, le compteur pourra être affiché tel quel.

On aurait pu placer dans cette partie d'initialisation la génération du nombre aléatoire. C'est une manière d'envisager les choses qui ne pose pas de problème. On peut considérer que ce travail fait partie de la préparation de la boucle ou au contraire qu'il est supposé fait quand on commence à envisager la boucle. C'est donc une affaire de goût, l'important est que le travail soit fait.

```
Initialisation
   iNombreEssais Devient 0
```

3. Cette formule est extraite du mode d'emploi de mon premier ordinateur, un TRS80 Pocket, qui disposait à l'époque d'une mémoire de 1024 octets et d'un processeur 4 bits. Il fonctionne encore aujourd'hui.

### Condition de déroulement

La condition de cette boucle est plus complexe que celle de notre premier exemple. Nous avons vu qu'il était indispensable qu'une boucle se termine. Dans notre premier exemple, nous jouions sur l'incrémentation d'un compteur, qui devait fatalement finir par atteindre une valeur maximale. Ici, notre boucle ne se terminerait normalement que quand le joueur aurait trouvé la solution du mystère. Si l'intervalle de nombres est grand et/ou la méthode du joueur anarchique, cet événement risque de ne jamais se produire. C'est la raison pour laquelle nous avons introduit un compteur d'essais. Le jeu se terminera quand le joueur aura trouvé la solution ou quand le nombre d'essais prévus sera épuisé.

Il ne peut y avoir qu'une seule condition dans une boucle, nous allons donc devoir composer cette condition à l'aide des deux sous-conditions qui viennent d'être énumérées et d'un connecteur logique. En fait, la boucle devra continuer si les deux conditions suivantes sont simultanément vraies :

1. le joueur n'a pas trouvé la solution (le contenu de la variable `iProposition` est différente de celui de la variable `iMystere`) ;
2. le nombre maximum d'essais n'est pas atteint, la variable `iNombreEssais` est inférieure à `ESSAIMAX`. Le fait que cette variable va s'incrémenter à chaque itération nous garantit bien que la boucle finira par se terminer.

Il nous faut combiner ces conditions à l'aide du connecteur `ET`, qui impose que les conditions soient simultanément vraies. Dès que l'une des deux sera fausse, la boucle s'arrêtera. Notons que les deux conditions peuvent être fausses en même temps, si le joueur trouve la solution au dernier essai.

L'écriture exacte de la condition dépendra de la valeur initiale du compteur d'essai et de la position de l'évaluation de la condition. Ce qui peut changer c'est la comparateur utilisé avec le `iNombreEssais`.

```
iProposition != iMystere ET iNombreEssais < ESSAIMAX
```

### Position du test

Dans ce cas, il n'y a pas à hésiter longtemps sur l'endroit où nous devons évaluer la condition. Nous ne pourrons dire que le joueur a trouvé ou non la solution qu'après l'avoir laissé jouer. De fait, le test portant sur la variable `iProposition`, le résultat n'aura de sens que quand cette variable aura été lue. Nous placerons donc l'évaluation à la fin de la boucle.

### Corps de la boucle

Le corps de la boucle comprend deux parties : l'interrogation du joueur et l'évaluation de sa solution. Les instructions ne présentent pas de difficulté particulière. Pour l'affichage des directives (trop grand ou trop petit), nous retrouvons en fait le problème vu au cours des exercices du précédent chapitre lors de la détermination d'un nombre positif, négatif ou nul, mais cette fois, nous comparons au nombre mystérieux au lieu de comparer à 0.

```
/* Lecture de la réponse */
Afficher "Essayez de deviner le nombre auquel je pense (essai "
   & iNombreEssais & ") : "
Lire iProposition

/* Traitement */
```

```
Si iProposition ?= iMystere Alors
    Afficher "Vous avez deviné."
Sinon /*Différent */
    Si iProposition < iMystere Alors
       Afficher "Le nombre est plus grand que " & iProposition
    Sinon /* Supérieur */
       Afficher "Le nombre est plus petit que " & iProposition
    Fin de Si
Fin de Si
Fin de Ligne
```

#### Moteur de la boucle

Nous avons vu plus haut que nous ne devons pas faire confiance à l'utilisateur pour voir se terminer la boucle. Il ne faut donc pas oublier d'incrémenter la variable `iNombreEssais`. De manière évidente, cette incrémentation doit se placer au début de la boucle si on veut pouvoir afficher le numéro de l'essai en cours (rappelons que nous partons de 0). Si on n'affiche pas le numéro, la place de cette instruction devient sans intérêt. Il ne faut cependant pas l'oublier et c'est une bonne idée de l'écrire tout de suite.

```
Moteur Incrémenter iNombreEssais
```

### 3.4.4 La fin du jeu

La fin du jeu consiste à donner le score et, éventuellement à afficher la solution pour celui qui ne l'a pas trouvée. Mais quand faut-il afficher la solution ? Une réponse qui vient rapidement à l'esprit consiste à tester la valeur de `iNombreEssais`. En faisant cela, nous risquons d'afficher un mauvais message si l'utilisateur trouve la solution au dernier essai. La seule méthode consiste donc à comparer à nouveau la proposition avec la solution [4].

```
Si iProposition ?= iMystere Alors
   Afficher "Vous avez réussi en " & iNombreEssais & " coups"
Sinon
   Afficher "Le nombre était " & iMystere
   & "\nVous ferez mieux une prochaine fois "
Fin de si
```

### 3.4.5 Programme complet

```
Constante ESSAIMAX = 15
Constante NOMBREMAX = 100
Variable Entière iMystere
Variable Entière iProposition
Variable Entière iNombreEssais

Début du traitement

/* Invention du nombre */
```

4. Nous verrons plus tard que répéter le dernier test peut parfois poser des problèmes, notamment avec les tableaux. Nous reviendrons plus tard sur cette question.

```
  Aléatoire
  iMystere <- HasardMax(NOMBREMAX) + 1
  Effacer Écran
  Initialisation
    iNombreEssais <- 0
  Répéter

    /* Lecture de la réponse */
    Moteur Incrémenter iNombreEssais
    Afficher "Essayez de deviner le nombre auquel je pense (essai "
      & iNombreEssais & ") : "
    Lire iProposition

    /* Traitement */
    Si iProposition ?= iMystere Alors
       Afficher "Vous avez deviné."
    Sinon /*Différent */
      Si iProposition < iMystere Alors
         Afficher "Le nombre est plus grand que " & iProposition
      Sinon /* Supérieur */
         Afficher "Le nombre est plus petit que " & iProposition
      Fin de Si
    Fin de Si
    Fin de Ligne
  Boucler tant que iProposition != iMystere ET iNombreEssais < ESSAIMAX
  Fin de Ligne
  Si iProposition == iMystere Alors
    Afficher "Vous avez réussi en " & iNombreEssais & " coups"
  Sinon
    Afficher "Le nombre était " & iMystere
    & "\nVous ferez mieux une prochaine fois "
  Fin de si
  Fin de Ligne
Fin du Traitement
```

## 3.5 Simplification des boucles : boucles avec compteur

Il arrive fréquemment qu'une boucle se contente de répéter des opérations identiques. Prenons l'exemple idiot d'un programme qui dit dix fois bonjour.

```
Variable Chaine sSalut
Variable Entière iIndice

Début du traitement
  Init
    iIndice <- 1
    sSalut <- "Bonjour"
  Tant que
    iIndice <=10
  Répéter
    Afficher sSalut
    Fin de Ligne
```

```
    Moteur Incrémenter iIndice
  Fin de Boucle

Fin du traitement
```

Ce type de boucle est tellement fréquent que la plupart des langages fournissent un mécanisme particulier qui permet de gérer cette boucle plus facilement :

- on désigne la variable compteur employée ;
- on donne sa valeur de départ et sa valeur finale
- le reste est implicite (le test en début de boucle, l'incrémentation en fin de boucle)

Nous laisserons quand même la partie initialisation de la boucle, qui pourra dans certains cas être vide, puisque le compteur s'initialise implicitement.

```
Variable Chaine sSalut
Variable Entière iIndice

Début du traitement

Initialisation
   sSalut <- "Bonjour"
Compter avec iIndice De 1 A 10 Croissant
   Afficher sSalut
   Fin de Ligne
Fin de Compter
Fin du traitement
```

`Croissant` s'oppose à `Décroissant` (pour parcourir la boucle en commençant par un nombre plus grand et en décrémentant). Par défaut, la boucle est supposée croissante. Insistons sur le fait que le test est supposé exécuté en début de boucle. Une boucle comme celle qui suit ne produira donc aucun affichage, puisque que la condition est fausse immédiatement.

```
Init
   iMax <- -7
   sSalut <- "Bonjour"
Compter avec iIndice De 1 A iMax Croissant
   Afficher sSalut
   Fin de Ligne
Fin de Compter
```

## 3.6 Choix d'un type de boucle

Lors de l'écriture d'une boucle, il importe de bien réfléchir à tous les composants que nous venons de voir : initialisation, condition, position de l'évaluation de la condition, corps de boucle et moteur. Cependant, les boucles avec compteurs permettent de simplifier l'écriture. Il faut donc en fait choisir tout d'abord le type de boucle :

- boucle avec compteur
- boucle avec test initial
- boucle avec test final

Il ne faut pas retenir par cœur les quelques exemples qui suivent. Il suffit de réfléchir un peu.

**les boucles avec compteur** s'emploieront toujours lorsqu'on peut dire, avant de commencer la boucle, combien de fois la boucle va s'exécuter (**N fois, N étant connu** [5]). Ce sera lorsqu'une variable contient le nombre d'itérations à effectuer :

- énumération comprise dans un intervalle de nombres
- nombre explicite d'opérations spécifié par l'utilisateur
- parcours complet d'un tableau (voir chapitre suivant)

**les boucles à test initial** s'emploieront lorsqu'on n'est pas certain d'effectuer au moins une fois la boucle (**O à N fois**). Cela signifie que la boucle ne va pas s'exécuter dans certains cas.

- opérations dépendant d'un contexte extérieur non connu (lecture des clients dans un fichier, qui peut être vide, affichage de données saisies qui peuvent ne pas exister...)
- parcours d'un tableau à la recherche d'un élément (on peut trouver directement)

**les boucles à test final** s'emploieront au contraire lorsqu'on est certain d'effectuer au moins une fois la boucle(**1 à N fois**). Ce sera souvent le cas lorsque la boucle opère une lecture de donnée qui détermine l'arrêt de la boucle :

- lecture d'un choix parmi plusieurs options
- continuation dépendant de la volonté de l'utilisateur.

Dans certains cas, le choix de la position n'a pas d'importance. Il entraînera cependant des modifications dans l'initialisation ou dans l'écriture du test.

# Exercices

**III-1.** Écrire un programme qui écrit les dix premiers nombres entiers.

**III-2.** Écrire un programme qui écrit les nombres entiers compris entre deux limites choisies par l'utilisateur.

**III-3.** Écrire un programme qui écrit le carré des dix premiers nombres entiers.

**III-4.** Rédiger un programme qui écrit les dix premiers nombres pairs.

**III-5.** Modifier le programme précédent pour qu'il imprime les N premiers nombres impairs.

**III-6.** Écrire un programme qui lit des nombres positifs entrés au clavier et en calcule la somme et la moyenne. On demandera à l'utilisateur combien de nombres il veut taper. Il faut tenir compte du cas où ce dernier ne veut rien taper.

**III-7.** Modifier le programme précédent de manière à ce que l'utilisateur ne soit plus obligé de déterminer à l'avance combien il va entrer de nombres. On va

5. On parle ici de la connaissance au moment de l'exécution. Les deux instructions suivantes sont donc équivalentes pour ce qui concerne le choix de notre type de boucle.

```
Initialisation
Lire iNombre
Compter avec iC de 1 à iNombre
...

Initialisation
iNombre<-10
Compter avec iC de 1 à iNombre
...
```

Dans le premier cas cependant, le programmeur ne connaît pas la valeur de `iNombre`.

utiliser une SENTINELLE pour indiquer au programme que la série est terminée. Une sentinelle est une valeur placée parmi les valeurs, mais reconnue comme une valeur témoin. On pourra par exemple ici choisir tout nombre négatif. Dès réception d'un nombre négatif, le programme s'arrêtera. Faites attention de ne pas additionner ce nombre avec les autres !

**III-8.** Écrire un programme qui teste si un nombre est premier (divisible seulement par lui-même). On va en fait tester le reste de la division par tous les nombres inférieurs au nombre donné (sauf 1). Remarquons qu'il existe deux raisons d'arrêter la recherche : quand on a trouvé un diviseur ou quand le diviseur devient trop grand. On aura sans doute besoin des opérateurs logiques `ET` et `OU`. Pour calculer le reste de la division, on utilise l'opérateur `%`. Par exemple, le reste de la division de 5 par 3 (qui vaut 2) s'écrira `5 % 3`. On prononce `%` « modulo »

**III-9.** Améliorer le programme précédent en tirant parti de la remarque suivante : s'il existe un grand nombre N qui divise un candidat nombre premier C, il existe forcément un petit nombre N' tel que N x N' = C . Passée une certaine limite, il devient donc inutile de continuer à examiner les diviseurs. Déterminez cette limite et optimisez votre programme.

**III-10.** Écrire un petit programme qui affiche un message quelconque et attende des réponses de l'utilisateur au clavier. La seule façon pour l'utilisateur d'arrêter le programme sera de taper un signe '$'.

**III-11.** Écrire un programme affichant un menu de tâches à réaliser. On va proposer quatre activités à l'utilisateur : manger, boire, danser ou dormir. L'utilisateur choisira l'activité de son choix (qui fera l'objet d'un simple message). Le programme reproposera ensuite le même menu jusqu'à ce que l'utilisateur choisisse d'arrêter. Exemple d'exécution :

```
Choisissez une activité :
1. Manger
2. Boire
3. Danser
4. Dormir
9. Terminer
2
Activité en cours : boire.
```

**III-12.** Reprenez le programme de l'équation et faites le tourner en boucle avec plusieurs jeux de données. Prévoir un dialogue qui demande à l'utilisateur s'il veut s'arrêter.

**III-13.** Le plus grand commun diviseur de deux nombres est le plus grand nombre qui les divise tous les deux exactement. Il existe plusieurs méthodes pour le déterminer. La plus compliquée consiste à décomposer chaque nombre en un produit de facteurs premiers, puis à prendre les facteurs communs.

$12 = 2 \times 2 \times 3$
$18 = 2 \times 3 \times 3$
$PGCD(12, 18) = 2 \times 3 = 6$

Il existe une autre méthode, plus facile. Elle part du principe que pour deux nombres A et B tels que A > B,

$$PGCD(A,\ B) = PGCD(B,\ A - B)$$

Il suffit donc de répéter l'opération un nombre suffisant de fois. Au bout d'un temps de calcul généralement fort court, on réduit le problème à chercher PGCD(N,N), ce qui est trivial et vaut N, ou PGCD(N,1), autre cas trivial. Rédiger un programme qui réalise cette tâche. Exemples :

$PGCD(18, 24) = PGCD(6, 18) = PGCD(12, 6) = PGCD(6, 6) = 6$

$PGCD(32, 44) = PGCD(12, 32) = PGCD(20, 12) = PGCD(8, 12) = PGCD(4, 8) = PGCD(4, 4) = 4$

$PGCD(7, 13) = PGCD(6, 7) = PCGCD(1, 6) = 1$

## Vocabulaire

ACCUMULATEUR, ARGUMENT D'UNE FONCTION, ALÉATOIRE (du latin ALEA, le dé), CONSTANTE, ÉLÉMENT NEUTRE, CONNECTEUR LOGIQUE, GÉNÉRATEUR DE NOMBRES ALÉATOIRES, INCRÉMENTER (DÉCRÉMENTER), ITÉRATION, SENTINELLE, TERME.