Initiation à l'algorithmique 5 premiers chapitres.

- Les articles -

Date de mise en ligne : mercredi 11 août 2004

Date de parution : 12 novembre 2003

Description :

Exemples, souvent très simples, qui ont pour but de présenter les principes de base de l'algorithmique de façon ludique.

Environnement iSeries

Maîtriser les principes de base de l'algorithmique permet de s'affranchir des contraintes des langages de programmation, et de concevoir des programmes bien écrits et aisément portables d'une plateforme à une autre. Un programme bien écrit est un programme facile à lire, à comprendre, à modifier et à corriger, même et surtout pour une personne étrangère à sa conception.

Cet article (que j'avais écrit il y a quelques années pour un club d'informatique, le GS Club) est composé d'exemples écrits en langage algorithmique, et certains de ces algorithmes sont traduits en Pascal (en TML Pascal plus exactement, qui était une variante du Pascal spécifique à l'Apple IIGS) ou en Basic. Ces exemples, souvent très simples, ont pour but de présenter les principes de base de l'algorithmique de façon ludique (je l'espère). J'essaie pour ma part d'appliquer ces principes de programmation dans mon travail, que j'écrive mes programmes en Adélia ou dans d'autres langages. Si vous souhaitez voir un exemple d'algorithme bien conçu, je vous invite à étudier le source de ma calculette AS/400, programme dont j'avais d'abord écrit l'algorithme avant de le coder en Adélia (peu d'effort en réalité, ce langage étant par nature algorithmique), puis en RPG (vous trouverez le source RPG dans ce site) sans plus de difficulté. Un autre exemple est le programme PGMPGM de Didier Encinas, dont il avait conçu l'algorithme avant de le coder en Adélia, je n'avais eu aucun mal à le réécrire en RPG.

Cet article avait été écrit avant tout pour des personnes débutant en programmation, mais j'invite les programmeurs expérimentés à le lire, histoire de vérifier s'ils sont aussi bons qu'ils le pensent. Cet article ne traite pas de programmation événementielle, ni de programmation orientée objets. Je me suis volontairement borné à des notions simples, que devraient maîtriser tout programmeur, quel que soit le type d'application qu'il développe (on se rend compte malheureusement qu'en pratique ce n'est pas toujours le cas, même chez des professionnels).

Et maintenant, si vous voulez bien me suivre pour les 5 premiers chapitres :

Chapitre 1 - Problèmes de permutation

Premier problème : Lire deux nombres A et B et échanger leur contenu.

Certains d'entre vous vont penser que je les prends pour des idiots et qu'il est évident qu'il faut utiliser une variable intermédiaire. A ceux-là je rétorquerai que ce n'est pas du tout évident et qu'il existe un moyen pour résoudre ce problème sans variable intermédiaire comme je le montre dans la solution n°2. Allez, avouez que ça vous en bouche un coin. Voici ma première solution :

Solution n°1 :

```
Variables :
 a, b, c : réels (c est la variable intermédiaire)
 Début
 Lire a, b
 c <-- a
 a <-- b
 b <-- c
 Afficher a, b
 Fin.
```

D'accord, ça ne ressemble pas à proprement parler à un programme en Pascal et si vous l'entrez tel quel sur votre compilateur, il plantera à coup sûr. Il s'agit d'un algorithme. Le langage Pascal ayant été créé par un professeur d'algorithmique, vous pouvez noter les similitudes avec le Pascal.

N.B. : le symbole "<—" est tout ce que j'ai trouvé pour symboliser une flèche avec le clavier. Ce symbole veut dire "reçoit". En d'autres termes, la variable c "reçoit" le contenu de la variable a, la variable a "reçoit" le contenu de la variable b, etc... Les basiqueurs doivent absolument oublier le "=" qui est une aberration, tant sur le plan logique que mathématique. En Pascal, la flèche se traduit par " :=".

En Pascal, il faut déclarer au préalable les variables que l'on va utiliser dans le corps du programme. Ce n'est pas une réelle contrainte. Cela oblige à une certaine rigueur, c'est tout. Cela permet également, lorsque l'on relit un programme 6 mois après l'avoir écrit, de s'y retrouver, surtout si l'on a pris la peine d'écrire son algorithme proprement et en le documentant correctement.

Voici le codage de l'algo en Pascal :

```
PROGRAM permutation (input, output) ;
 VAR
 a, b, c : REAL ;
 BEGIN
 WRITE (' Introduisez le premier nombre ') ;
 READLN (a) ;
 WRITE (' Introduisez le deuxieme nombre ') ;
 READLN (b) ;
 c := a ;
 a := b ;
 b := c ;
 WRITELN (a:5:2,' ',b:5:2) ;
 WRITELN (' Pressez RETURN pour quitter ') ;
 READLN ;
 END.
```

N.B. : WRITE et WRITELN sont équivalents au PRINT du Basic et READLN à INPUT. En Pascal, les guillemets du Basic sont remplacés par des apostrophes pour afficher les chaines de caractères. Les parenthèses sont indispensables, ainsi que les points-vigules. Lors de l'affichage de a et de b à la fin du programme, vous notez que j'ai ajouté ":5:2" après a et b, ce qui veut dire que la machine affichera le contenu de a et le contenu de b avec 5 chiffres avant la virgule et 2 chiffres après. C'est tout à fait facultatif.

J'utilise le mode Input,Output du TML/Pascal qui est bien pratique pour débuter car il permet de travailler dans un

environnement en mode texte. Le READLN à l'avant-dernière ligne nécessite que l'utilisateur presse la touche RETURN sinon il n'aurait pas le temps de lire le résultat de la permutation.

Tapez ce programme, exécutez-le, vérifiez que la permutation s'effectue bien quelles que soient les valeurs de a et de b (nombres entiers, à virgule ou négatifs). Comparez l'algorithme et son programme Pascal. Vous noterez que dans l'algorithme je ne perds pas de temps à gérer des affichages du style "Introduisez le premier nombre", ces problèmes bassement terre-à-terre étant réservés à la rédaction dans un langage. Vous noterez également qu'un algorithme comme celui-là est très facilement transposable dans n'importe quel langage (Basic, Cobol, Pascal, etc...).

Et maintenant, chose promise, chose dûe, voici ma...

...Solution n°2 (sans variable intermédiaire. Mais si, c'est possible) :

```
Variables :
 a, b : réels
 Début
 Lire a, b
 b <-- b + a
 a <-- b - a
 b <-- b - a
 Afficher a, b
 Fin.
```

Surtout ne vous laissez pas impressionner. On ne se sert jamais de cette solution même si elle est très belle sur le plan de la logique. Elle a néanmoins une grande valeur pédagogique car elle permet au débutant de bien comprendre l'utilisation qui peut être faite des variables. Pour vous aider à décortiquer cette solution, commencez par la coder en Pascal. Puis faisons ensemble son "jeu d'essai", que l'on appelle aussi "table de vérité". Cela consiste à tester dans un tableau sommaire les valeurs successives des variables utilisées, en suivant pas à pas le cheminement du programme pour envisager tous les cas de figure et s'assurer qu'ils fonctionnent :

```
 a  !  b
 -----!------
 3  !  9 ( Lire a, b )
 3  ! 12 ( b <-- b + a )
 9  ! 12 ( a <-- b - a )
 9  !  3 ( b <-- b - a )
 9  !  3 ( Afficher a, b )
```

Entrainez-vous à faire ce jeu d'essai avec d'autres valeurs telles que des nombres à virgule ou négatifs. Faites aussi le jeu d'essai de la solution n°1, histoire de vous faire la main. Faites également le jeu d'essai de votre propre solution et, si elle ne marche pas, tâchez de voir pourquoi et comment la corriger ou l'améliorer.

Pour ceux qui se poseraient des questions quant à l'utilité des permutations, disons qu'elles sont le plus souvent utilisées pour effectuer des transferts de valeurs d'un tableau à l'autre ou dans un même tableau. Vous verrez les notions de tableau plus tard, rien ne presse.

Je vais vous soumettre un autre problème que je laisserai à votre appréciation. En vous référant à la solution n°1

(avec variable intermédiaire) vous devriez le résoudre assez facilement. Lorsque ce sera fait, essayez de le résoudre avec la solution n°2 (sans variable intermédiaire) : Lire trois nombres a, b et c et réalisez une permutation circulaire. En d'autres termes :

a —> b —> c —> a

Si vous séchez, utilisez une variable intermédiaire d pour effectuer la permutation suivante :

a —> b —> c —> d —> a

Là, j'ai peur d'en avoir trop dit.
Chapitre 2 - Structures alternatives

Vous êtes remis de vos émotions ? Très bien. Nous allons maintenant étudier les structures dites alternatives du genre :

```
Si <condition>
 |  Alors <action1>
 |  Sinon <action2>
 FinSi
```

En d'autres termes : **Si** la condition est remplie **Alors** exécuter l'action1 **Sinon** exécuter l'action2.

Exemple :

```
Si <il pleut>
 |  Alors <je prends mon parapluie>
 |  Sinon <je ne prends pas mon parapluie>
 FinSi
```

En langage courant on appelle ça un test. Vous verrez, ça n'a rien d'effrayant et c'est très pratique.

Vous noterez que j'ai ajouté des traits verticaux pour améliorer la lisibilité de l'algorithme. C'est facultatif, mais vous verrez que cela devient quasiment indispensable quand vous commencerez à imbriquer des tests ce qui ne saurait tarder.

Problème n°1 :

Lire trois variables a, b, c quelconques. Permuter le contenu de ces variables de façon à obtenir a >= b >= c. Ecrivez l'algo, le jeu d'essai, puis le programme Pascal correspondant. (Eh oui, ça se corse, mais il faut bien en passer par là). Réfléchissez un moment au problème puis revenez à cet article.

Dans le cas présent, trois tests (IF..THEN..) suffisent. Il est évident que l'on va se servir ici des permutations vues précédemment. Maintenant que vous savez tout ça, retournez à votre solution et réfléchissez au moyen de l'améliorer. Si vous ne trouvez pas du premier coup, ne vous acharnez pas inutilement. Soufflez un peu et revenez-y. En programmation, on progresse par palier : on peut buter pendant un certain temps sur un problème, puis brusquement tout semble s'éclaircir comme par miracle. Chaque personne a son propre rythme d'apprentissage, c'est encore plus vrai pour les programmeurs.

Passons aux choses sérieuses. Avant de me lancer dans la rédaction de l'algo, je vais, pour vous et en excluvisité, faire une analyse du problème pour cerner tous les pièges à éviter.

Commençons par passer en revue tous les cas de figure :

```
Cas  <Condition>        <Action>
 1 - Si a > b > c Alors Rien
 2 - Si a > c > b Alors Permuter b et c
 3 - Si b > a > c Alors Permuter a et b
 4 - Si b > c > a Alors Permuter a et b puis b et c
 5 - Si c > a > b Alors Permuter b et c puis a et b
 6 - Si c > b > a Alors Permuter a et c
```

On remarque que "Permuter b et c" se répète dans 3 cas.
On remarque que "Permuter a et b" se répète dans 3 cas.
On remarque que "Permuter a et c" ne se présente qu'une fois.

Ces 6 hypothèses se résument donc à 3 cas maximum. On doit donc pouvoir réduire le problème à 3 conditions :

```
1 - Si c > a Alors Permuter a et c
 2 - Si c > b Alors Permuter b et c
 3 - Si b > a Alors Permuter a et b
```

Voilà, le plus gros du travail est fait.

Algorithme :

```
Variables :
 a, b, c, d : réels (d est la variable intermédiaire)
 Début
 Lire a, b, c
 Si c > a
 |  Alors Faire Permuter (a,c)
 FinSi
 Si c > b
 |  Alors Faire Permuter (b,c)
 FinSi
 Si b > a
 |  Alors Faire Permuter (a,b)
 FinSi
 Afficher a, b, c
 Fin.
```

Sans cet effort d'analyse, je ne serais sans doute jamais parvenu à une telle simplification du problème. J'ai volontairement simplifié la rédaction de l'algo au niveau des actions pour ne pas l'alourdir. Vous remarquerez qu'un algo comme celui-ci ressemble étrangement à du langage naturel. La traduction en Pascal ressemble plus à un exercice de traduction que de programmation. Au moins, il ne s'agit plus ici de bidouille, mais presque véritablement de programmation rigoureuse et efficace. Avant de passer au codage en Pascal, faisons quelques jeux d'essai pour s'assurer de la bonne marche du programme.

```
Jeu d'essai n° 1 :
```

```
 a  !  b  !  c
 -----!-----!-------
 1  !  5  !  2
 2  !  5  !  1
 5  !  2  !  1
Jeu d'essai n° 2 :
 a  !  b  !  c
 -----!-----!-------
 9  !  4  !  5
 9  !  5  !  4
```

Les 2 exemples ci-dessus correspondent respectivement aux cas n°5 et n°2. Vérifiez pour les 4 autres cas que le programme fonctionne correctement.

Passons au codage en Pascal :

```
PROGRAM permutation (input, output) ;
 VAR
 a, b, c, d : REAL ;
 BEGIN
 WRITE (' Introduisez le premier nombre ') ;
 READLN (a) ;
 WRITE (' Introduisez le deuxieme nombre ') ;
 READLN (b) ;
 WRITE (' Introduisez le troisieme nombre ') ;
 READLN (c) ;
 IF c >= a THEN
 BEGIN
 d := a ;
 a := c ;
 c := d ;
 END ;
 IF b >= a THEN
 BEGIN
 d := a ;
 a := b ;
 b := d ;
 END ;
 IF c >= b THEN
 BEGIN
 d := c ;
 c := b ;
 b := d ;
 END ;
 WRITELN (a,' ',b,' ',c) ;
 READLN ;
 END.
```

Comparez la syntaxe de l'algo et du Pascal pour vous familiariser avec leurs différences. Notez que le FinSi de l'algo se traduit par un " ;" en Pascal.

Autre exemple d'algorithme :

```
Lire Note_candidat
 Si Note_candidat < 8
 |  Alors
 |  |  le candidat est éliminé
 |  Sinon
 |  |  Si Note_candidat < 10 ( 8 <= note < 10 )
 |  |  |  Alors le candidat est admis à l'épreuve de rattrapage,
 |  |  |  Sinon le candidat est reçu ( note >= 10 )
 |  |  FinSi
 |  FinSi
 FinSi
```

Cet exemple n'offre pas un grand intérêt en soi mais il peut constituer un sous-ensemble d'une application de gestion de candidatures. Nous reverrons cela avec les procédures.

P.S. : pour ne pas vous laisser sur votre.. faim ?, voici le dessert : un petit problème supplémentaire et ma solution : Lire une chaîne de caractères, la partager en deux, centrer et afficher les deux moitiés à l'écran en les accolant. Voilà l'occasion de commencer à manipuler les chaînes de caractères.

Algo simplifié (je vous laisse le soin d'écrire l'algo détaillé) :

```
Début
 1 - lire la chaine ;
 2 - récupérer sa longueur (avec l'instruction LENGTH) ;
 3 - sachant que l'écran a 80 caractères de large, calculer l'espace à
 insérer entre le bord gauche de l'écran et la chaine pour qu'elle
 soit centrée à l'écran ;
 4 - recopier dans moitie1 la première moitié de la chaîne (avec COPY) ;
 5 - recopier dans moitie2 la seconde moitié de la chaîne (avec COPY) ;
 6 - afficher moitie1 et moitie2 accolées et centrées ;
 Fin.
```

Programme Pascal :

```
PROGRAM texte (input, output) ;
 USES consoleIO ;
 VAR
 phrase : STRING[80] ;
 moitie1, moitie2 : STRING[40] ;
 longueur, espace, demi_phrase : INTEGER ;
 BEGIN
 WRITELN ('Entrez votre phrase et pressez RETURN.') ;
 READLN (phrase) ;
 longueur := LENGTH (phrase) ;
 espace := ( 80 - longueur ) DIV 2 ;
 demi_phrase := long DIV 2 ;
 moitie1 := COPY (phrase, 1, demi_phrase) ;
 moitie2 := COPY (phrase, demi_phrase+1, demi_phrase) ;
 EraseScreen ;
 GotoXY (espace, 10) ;
 WRITE (moitie1) ;
 WRITELN (moitie2) ;
```

```
REPEAT UNTIL KEYPRESSED ;
END.
```

Là encore, voici une proposition de solution. Ce n'est pas forcément la meilleure. A vous de voir. Certaines instructions comme ConsoleIO (gestion des interfaces avec le poste de travail), LENGTH (pour récupérer la longueur d'une chaîne de caractères), COPY (copie d'une chaîne de caractères) ou REPEAT UNTIL KEYPRESSED (boucler jusqu'à ce qu'une touche soit pressée) trouvent leurs équivalents dans d'autres langages que le Pascal . Bon amusement.

Chapitre 3 - Les structures répétitives

Avant de passer à la suite de ce cours, il me semble opportun de rappeler comment se conçoit un algo. Lorsqu'on lit le sujet d'un problème de programmation, on doit commencer par se demander quels problèmes sont posés et quels outils on devra utiliser pour les résoudre. La meilleure solution, en fait, consiste à poser sur le papier les différents cas de figure possibles, à concevoir une première ébauche d'algorithme et à vérifier "à la main" qu'il fonctionne dans tous les cas de figure (rappelez-vous le problème de la permutation circulaire). Evidemment ça c'est la théorie et en fait tout dépend de la complexité du problème posé. Le choix du langage pourra se faire plus tard, en fonction des besoins. A propos de langage, le Pascal a comme avantage d'être bien adapté à l'apprentissage de l'algorithmique et de la programmation structurée. C'est normal, il a été conçu par Nicklaus Wirth, un professeur d'algorithmique. Un algo bien fait ne doit donc pas poser de problème d'adaptation, que ce soit en Pascal ou dans d'autres langages tels que Visual Basic.

A titre d'information, le dictionnaire de l'informatique aux éditions Larousse donne deux définitions de l'algorithmique :

1. Science des algorithmes utilisés en informatique ;
2. Connaissance des méthodes utilisées pour construire des algorithmes et des programmes.

Vous remarquerez au fil des exemples que je n'emploie jamais dans mes algos d'instruction de branchement de type GOTO. En algorithmique on n'en a pas besoin. Les programmes "spaghetti" ou "jeu de piste" ça vous dit quelque chose ? Certains programmeurs RPG en sont des spécialistes. Si c'est le cas de vos programmes RPG, Basic ou autres, je ne saurais trop vous conseiller de lire ce cours en entier et d'étudier attentivement tous les exemples. S'il n'a pas la prétention de faire de vous des programmeurs chevronnés, il vous permettra, du moins je l'espère, d'améliorer la logique de vos propres programmes, de progresser dans votre découverte de la programmation et de manipuler des concepts qui peuvent paraître obscurs au premier abord. Les exercices proposés dans ce cours sont fournis avec des exemples de solution (j'ai bien dit des exemples, prenez-les pour ce qu'ils sont et n'hésitez pas à les améliorer). Une dernière mise au point : je donnerai désormais rarement l'équivalent Pascal des algos présentés ici. Moi ça me fera moins de travail, et vous ça vous fera un peu d'entraînement, en vous laissant de plus le choix du langage utilisé.

Mais commençons par le commencement.

Qu'est-ce qu'une boucle et à quoi ça sert ?

Les boucles sont des structures répétitives qui permettent de répéter un ensemble d'actions un nombre de fois déterminé soit par l'utilisateur, soit par le programmeur, soit par un calcul effectué par le programme. Cela permet au programmeur de s'épargner bien du travail. Etudions la syntaxe algorithmique d'une boucle dans un exemple.

N.B. : "TQ" et "FTQ" signifient respectivement "Tant Que" et "Fin de Tant Que". "FSi" signifie "Fin de Si". Lire se traduit par READLN en Pascal (ou INPUT en Basic) et Afficher se traduit par WRITELN en Pascal (ou PRINT en Basic). "Si..Alors..Sinon..FSi" se traduit par "IF..THEN..ELSE.. ;" en Pascal. En Basic, la syntaxe peut varier selon

les versions et certains vieux Basic n'ont pas le "ELSE" ce qui oblige à feinter avec un "GOTO". La flêche "<—" se dit "reçoit" et se traduit par := en Pascal et par = en Basic. De toute façon, reportez-vous au manuel de votre langage pour les problèmes de syntaxe. N'hésitez pas, en algorithmique et en Pascal, à prendre des noms de variables explicites pour améliorer la lisibilité de vos programmes.

Problème 1 : programme de devinette d'un nombre (un grand classique !). Il devra indiquer si le nombre donné par l'utilisateur est trop grand ou trop petit et afficher le nombre de coups en fin de partie.

```
Lexique
 gagne : nombre pris au hasard entre 1 et 100
 n : variable de type nombre entier (sert à compter le nombre de coups)
 nombre_lu : variable de type entier (demandée au joueur par programme)
 Début
 Lire nombre_lu
 n <-- 1
 TQ nombre_lu <> gagne
 |  Si nombre_lu > gagne
 |  |  Alors Afficher 'Trop grand !'
 |  |  Sinon Afficher 'Trop petit !'
 |  FSi
 |  n <-- n + 1
 |  Lire nombre_lu
 FTQ
 Afficher 'Vous avez trouvé en ',n,' coup(s).'
 Fin.
```

Le programme de devinette est classique mais aussi très didactique. Faites son jeu d'essai pour vous aider à comprendre son déroulement avec un tableau du type :

```
 nbre_lu   !   gagne   !   n   !   Message affiché
 ------------!-----------!-------!---------------------
 !           !       !
```

La boucle se répète donc Tant Que (While en anglais) la condition n'est pas remplie, c'est à dire tant que "nbre_lu" est différent de "gagne". Vous remarquerez que si par chance le joueur trouve le contenu de "gagne" du premier coup, alors le programme n'entre pas dans la boucle et passe directement à l'avant dernière ligne pour afficher le nombre de coups. D'où l'importance d'initialiser n à 1 au départ. n est une variable qui sert de compteur. Il est judicieux de la déclarer en entier (INTEGER en Pascal). Vous remarquerez par la suite qu'on pouvait trouver une solution plus jolie, mais je préfère démarrer doucement.

Avant de poursuivre, il convient de présenter les différentes façons de gérer les boucles en Pascal. L'algorithme suivant :

```
increment <-- 1
 TQ increment < 13
 |  Afficher increment
 |  increment <-- increment + 1
 FTQ
```

Pourra s'écrire :

- 1ère solution :

```
increment := 1 ;
 WHILE increment < 13 DO
 BEGIN
 WRITELN(increment) ;
 increment := increment + 1 ;
 END ;
```

- 2ème solution :

```
increment := 1 ;
 REPEAT
 WRITELN(increment) ;
 increment := increment + 1 ;
 UNTIL increment >= 13 (Répète ... Jusqu'à ...)
```

- 3ème solution :

```
FOR increment := 1 TO 12 DO
 BEGIN
 WRITELN(increment) ;
 END ;
```

N.B. : Cette dernière solution n'est possible ici que parce que l'incrémentation se fait de 1 en 1. Il faut remarquer qu'à la sortie de la boucle, la variable "increment" aura la valeur 12, tandis que dans les 2 premiers cas elle sera à 13. Vous pouvez le vérifier en ajoutant un WRITELN(increment) après la boucle. Il n'existe pas en Pascal d'instruction STEP comme en Basic, ce qui oblige à recourir à l'une des deux premières solutions lorsque le pas est différent de 1. Vous remarquerez aussi que si vous utilisez REPEAT UNTIL plutôt que WHILE DO il sera nécessaire d'inverser la condition de sortie de la boucle. Faites des essais pour bien comprendre la subtilité de la chose. A noter qu'en GS.Basic, il existe la structure WHILE UNTIL équivalente au REPEAT UNTIL du Pascal. Notez aussi que le pas peut tout aussi bien être négatif. En Pascal, on pourra aussi utiliser la structure FOR..DOWNTO..DO.. si le pas est de -1.

Petite astuce :

```
increment := increment + 1 ;
```

peut s'écrire en Pascal :

```
INC(increment) ;
```

INC est une fonction qui incrémente l'argument entre parenthèses de 1. Il existe la fonction DEC qui, elle, décrémente de 1. Nous reparlerons des fonctions dans le prochain cours et nous apprendrons même à en créer.

Attention : WHILE DO n'est pas équivalent à REPEAT UNTIL, car avec REPEAT UNTIL, si la condition de sortie de la boucle est vraie dès le départ, celle-ci est quand même parcourue une fois, ce qui n'est pas le cas avec WHILE DO. Préférez donc l'utilisation de WHILE DO, ou de FOR TO DO, vous éviterez les mauvaises surprises. Mais je vous recommande d'expérimenter ces différentes solutions afin de vous familiariser avec leurs différences.

N.B. : l'initialisation des variables en début de programme ou avant une boucle n'est pas obligatoire. Tout dépend en

fait du contexte et des besoins du moment.
Chapitre 4 - Les booléens

Problème 1 :

Ecrire un programme demandant à l'utilisateur s'il aime le choux-farci jusqu'à ce que ce dernier réponde oui. Voilà l'occasion de vous présenter les booléens.

```
Lexique
 bonne_reponse = "O" (constante, réponse que le programme attend)
 reponse_lue : variable de type caractère (CHAR en Pascal)
 encore : variable de type booléen (BOOLEAN en Pascal)
 Début
 encore <-- vrai (en Pascal on écrirait: encore := TRUE)
 TQ encore
 |  Afficher 'Aimez-vous le choux-farci ?'
 |  Lire reponse_lue
 |  encore <-- reponse_lue <> bonne_reponse
 FTQ
 Afficher 'Vous y avez mis le temps mais vous avouez enfin.'
 Fin.
```

Un booléen est une variable qui peut prendre 2 états : VRAI ou FAUX (TRUE or FALSE). Ce type de variable existe en Pascal mais pas en Basic, ce qui est dommage car il rend de grands services.

Au démarrage "encore" reçoit VRAI. Le programme entre donc dans la boucle. Si l'utilisateur entre une réponse différente de "O", alors "reponse" recevra encore la valeur VRAI. Le programme exécute à nouveau la boucle et repose à nouveau la même question (complètement idiote, je l'admets). Si cette fois l'utilisateur entre la bonne réponse (en l'occurrence "O"), alors la comparaison entre "reponse_lue" et "bonne_reponse" retourne la valeur FAUX dans la variable "encore". La boucle ne s'exécutant que Tant Que "encore" = VRAI, le programme sort de la boucle pour exécuter la fin du programme.

Je sais, l'emploi des booléens peut paraître obscur et la bonne connaissance de l'algèbre booléenne n'est pas superflue. Il faudrait encore beaucoup d'exemples de ce type pour vous familiariser avec les booléens. N'hésitez pas à vous en écrire, à les faire "tourner" par jeu d'essai interposé, à les écrire en Pascal. Il est d'autant plus malaisé pour moi de vous expliquer ça par tube cathodique interposé, mais je ne pouvais décemment pas éluder la question. Il est nécessaire de bien comprendre qu'un booléen est tout bonnement un test qui s'effectue sous forme d'un calcul "logique" et qu'on pourrait très bien le remplacer par un IF..THEN.. classique. Mais le booléen a le mérite de clarifier les programmes, d'être peu gourmand en mémoire et en temps machine. Il est parfois incontournable sur certains problèmes. Essayez pour vous "faire la main" de remplacer le test du premier exemple par un booléen et de réécrire le second exemple sans booléen. Il y aurait encore beaucoup à dire. Je têacherai d'y revenir dans les prochains exemples.

Problème 2 :

Saisir les notes d'un élève les unes à la suite des autres, faire ensuite le cumul des notes et calculer la moyenne.

```
Lexique
 cumul : variable de type réel
 note_lue : variable de type réel
 nbre_notes : variable de type entier (encore un "compteur")
 Début
 cumul <-- 0
 nbre_notes <-- 0
 Lire note_lue
 TQ note_lue <> -1
 |  cumul <-- cumul + note_lue
 |  nbre_notes <-- nbre_notes + 1
 |  Lire note_lue
 FTQ
 Si nbre_notes = 0
 |  Alors Afficher 'Pas de notes, fin de traitement'
 |  Sinon Afficher 'Moyenne de l''élève : ', cumul / nbre_notes
 FSi
 Fin.
```

Explications : Le programme initialise "cumul" à 0 et "nbre_notes" à 0 également. Le programme demande ensuite à l'utilisateur d'entrer la première note. Lorsqu'il désire arrêter la saisie, il lui suffit d'entrer "-1" qui est ici la condition de sortie de la boucle de saisie. Le programme passe alors à l'édition de la moyenne de l'élève. Pour bien comprendre, faisons quelques jeux d'essai (les nombres dans les tableaux ci-dessous sont en fait le contenu des variable au fur et à mesure du déroulement du programme) :

```
 nbre_notes ! cumul ! note_lue ! Moyenne
 ------------!-------!----------!------------
 0     !   0   !    15    !
 1     !  15   !    25    !
 2     !  40   !    -1    !   20
```

Voyons un second jeu d'essai :

```
 nbre_notes ! cumul ! note_lue !     Moyenne
 ------------!-------!----------!--------------------
 0     !   0   !    -1    ! Pas de notes, fin du programme
```

En effet si l'utilisateur, pour une raison quelconque, veut sortir du programme sans avoir entré de note, il entraine la division de "cumul" (qui est à 0) par "nbre_notes" (qui est aussià 0). Chacun sait que la division par 0 est impossible, l'ordinateur sait faire des choses idiotes mais pas celle-là. Donc, pour éviter un plantage lors de l'affichage de la moyenne, j'ai ajouté un test qui traite le cas où on n'entre pas dans la boucle.

Il y en a certainement qui vont me répliquer qu'on aurait pu faire plus joli. Par exemple en demandant à l'utilisateur après chaque note s'il veut en ajouter une autre avant le calcul de la moyenne. C'est très joli en effet, mais allez dire ça au malheureux utilisateur qui doit entrer les notes de plusieurs dizaines d'élèves pour le traitement d'un examen par exemple. S'il a dû appuyer 1200 fois sur la touche "O" (pour Oui) pour entrer 1200 notes, mieux vaut pour vous ne pas le croiser en fin de journée. J'ai donc pris comme condition de sortie "-1" mais c'est tout à fait arbitraire et j'aurais pu choisir autre chose (9999 par exemple) puisque les notes s'échelonnent d'habitude entre 0 et 20. Dans un programme plus sophistiqué faisant appel à une interface utilisateur plus conviviale, on pourrait programmer la sortie par une touche de fonction ou un clic sur un bouton, mais ça c'est une autre histoire.

Problème 3 :

Ecrire un programme qui demande à l'utilisateur d'entrer une série de nombres et indique en fin de programme le nombre le plus grand de la série. A mon avis, si vous avez bien compris ce qui précède, cet exercice ne devrait pas vous poser de problème. Essayez quand même de le résoudre avant de passer à la lecture de ce qui suit (faites aussi le jeu d'essai).

```
Lexique
 nombre_lu : variable de type nombre réel (REAL en PASCAL)
 plus_grand : variable de type réel (stockage du plus grand nombre)
 Début
 plus_grand <-- 0
 Afficher 'Entrez votre nombre '
 Lire nombre_lu
 TQ nombre_lu <> -1 (-1 est la condition d'arrêt)
 |  Si nombre_lu > plus_grand
 |  |  Alors plus_grand <-- nombre_lu
 |  FSi
 |  Afficher 'Entrez votre nombre'
 |  Lire nombre_lu
 FTQ
 Afficher 'Le plus grand nombre est : ', plus_grand
 Fin.
```

Le programme, après avoir initialisé "plus_grand" à 0, demande à l'utilisateur d'entrer un nombre. Si "nombre_lu" est différent de -1, le programme entre dans la boucle. Il compare alors le contenu de la variable "nombre_lu" avec celui de la variable "plus_grand". Si "nombre_lu" est supérieur à "plus_grand" alors le contenu de "plus_grand" est modifié pour recevoir celui de "nombre_lu". Mais si "nombre_lu" est inférieur ou égal à "plus_grand" la mise à jour de "plus_grand" est inutile. Le programme ignore alors ce qui se trouve après le "Alors" du test et passe directement à la saisie d'un nouveau nombre. Il reteste le contenu de "nombre_lu" au niveau du TQ et si "nombre_lu" est égal à -1, alors le programme sort de la boucle.

Maintenant que vous savez tout ça, modifiez l'algo numéro 2 pour qu'il soit capable de vous donner, en plus de la moyenne de l'élève, sa note la plus haute, et pourquoi pas aussi sa note la plus basse. Je vous laisse vous débrouiller. Avec ce que vous savez, ça ne devrait pas vous poser de problème.

Problème 4 :

Ecrire un programme qui lit une phrase terminée obligatoirement par un point. Il devra indiquer le nombre de mots (sachant que les mots sont séparés par au moins un espace et qu'il n'y a pas d'espace entre le dernier mot et le point), le nombre de majuscules et de caractères autres. Accrochez-vous car ça se corse. Ce programme va utiliser 3 boucles dont 2 imbriquées dans une troisième, ainsi que des tests, des variables servant aux cumuls et, le plus important, une variable de type CHAINE (STRING en Pascal) qui va nous permettre de nous familiariser avec les tableaux, une chaêine de caractères pouvant être considérée comme un tableau de caractères et fonctionnant à peu près de la même façon. Je suis conscient de la difficulté de l'exercice et vais essayer de bien expliquer l'algo. Mais vous verrez, ce n'est pas si terrible que ça.

```
Lexique
```

```
i : variable de type entier (va servir de compteur)
mots : variable de type entier (va servir au cumul du nombre de mots)
majus : variable de type entier (pour cumul du nombre de majuscules)
autre : idem (cumul du nombre de caractères autres : *az10-3"é'.)
phrase : variable de type chaine

Début
Afficher 'Entrez votre chaîne (pas plus de 80 caractères SVP)'
Lire phrase
mots <-- 0 (initialisation des variables de cumul)
majus <-- 0
autre <-- 0
i <-- 1 * compteur à 1 pour lecture premier caractère de "phrase" *
********** BOUCLE 1 **********
TQ phrase(i) <> '.'
|  ********* BOUCLE 2 *********
|  TQ phrase(i) <> '.' ET phrase(i) <> ' '
|  |  Si phrase(i) >= 'A' ET phrase(i) <= 'Z'
|  |  |  Alors majus <-- majus + 1
|  |  |  Sinon autre <-- autre + 1
|  |  FSi
|  |  i <-- i + 1
|  FTQ
|  ****** Fin de boucle 2 ******
|  ******** BOUCLE 3 ********
|  TQ phrase(i) = ' '
|  |  i <-- i + 1
|  FTQ
|  ****** Fin de boucle 3 ******
|  mots <-- mots + 1
FTQ
****** Fin de boucle 1 ******
Afficher 'Nombre de mots : ', mots
Afficher 'Nombre de majuscules : ', majus
Afficher 'Nombre de caractères autres : ', autre
Fin.
```

Explications : je crois que vous les avez bien méritées.

Le programme commence par demander à l'utilisateur d'entrer une chaîne terminée obligatoirement par un point, ce dernier servant au programme pour "trouver" la fin de la chaîne. Puis il met à 0 les variables servant au cumul du nombre de mots, de majuscules, et de caractères autres. Pour bien comprendre la suite du programme, prenons un exemple :

Supposons que l'utilisateur entre la chaîne suivante :

```
A bientot.
```

La variable i qui sert à se "déplacer" dans la chaîne est mise à 1 pour la lecture du premier caractère de la chaîne. Le programme teste s'il peut entrer dans la Boucle 1 en comparant le premier caractère avec l'indicateur de fin de chaîne qui est un point. Puisque le premier caractère est différent d'un point, le programme entre donc dans la

Boucle 1. Il teste ensuite s'il peut entrer dans la Boucle 2, boucle qui a pour but de calculer le nombre de majuscules et de caractères autres. Pour entrer dans cette boucle, il doit être différent d'un point, et aussi d'un espace, car n'oublions pas que nous devons aussi compter le nombre de mots, les mots étant séparés par au moins un espace. Notre premier caractère répond favorablement aux conditions requises, donc on entre dans la boucle 2. Dans cette boucle, on teste si le premier caractère de la chaîne est supérieur ou égal à 'A' et inférieur ou égal à 'Z'. Il est important de comprendre ici que le programme teste en fait les valeurs ASCII des caractères 'A' et 'Z' et les compare à la valeur ASCII du premier caractère de la chaîne. Dans le cas présent, le premier caractère de la chaîne est égal au caractère 'A' donc le programme ajoute 1 à la variable de cumul "majus" et ignore le Sinon pour passer à la suite de la Boucle 2. La variable 'i' est incrémentée de 1 et on retourne au TQ de la boucle 2. Cette fois, le second caractère de la chaîne est un espace donc le programme quitte la Boucle 2 et passe à la Boucle 3. Cette troisième boucle a pour but d'éliminer tous les espaces superflus entre les mots. Dans le cas présent il n'y a qu'un espace donc la Boucle 3 ne tourne qu'une fois avant de s'arrêter. Le programme passe alors au cumul du nombre de mots puis on retourne au TQ de la première boucle et rebelotte. Je vous laisse étudier le programme plus en détail. Lorsque vous l'aurez bien compris, tâchez de le modifier pour qu'il affiche également le nombre de minuscules et de chiffres.

Si vous manquez d'idées, voici un exemple d'application de ce que nous avons vu : programme comptant le nombre de mots d'une chaîne en utilisant un booléen appelé "mot" pour tester si on se trouve dans un mot ou non.

Ou encore : programme qui donne la longueur du mot le plus long d'une phrase terminée par un point. Chaque mot est séparé par un ou plusieurs espaces.

Problème 5 :

Ecrire un programme lisant 2 nombres entrés par l'utilisateur et lui permettant de choisir l'une des 4 opérations suivantes + - / * pour calculer ces 2 nombres. Pour corser un peu le problème, disons que l'utilisateur ne pourra sortir du programme qu'en entrant le caractère "Q" (pour Quitter). Voilà une occasion supplémentaire d'utiliser un booléen.

```
Lexique :
 nombre1, nombre2 : variables de type réel (REAL en Pascal)
 sortir : booléen (BOOLEAN en Pascal)
 choix : variable de type caractère (CHAR en Pascal)

 Début
 Lire nombre1, nombre2
 sortir <-- FAUX
 Afficher 'Entrez "Q" pour Quitter'
 TQ non sortir
 Afficher 'Quelle opération voulez-vous ? (+-*/) '
 Lire choix
 Si choix = 'Q' Ou choix = 'q'
 Alors sortir <-- VRAI
 Sinon
 Si choix = '+'
 Alors Afficher nombre1 + nombre2
 Sinon
 Si choix = '-'
 Alors Afficher nombre1 - nombre2
```

```
 Sinon
 Si choix = '*'
 Alors Afficher nombre1 * nombre2
 Sinon
 Si choix = '/'
 Alors Afficher nombre1 / nombre2
 Sinon Afficher 'Erreur dans votre choix'
 FSi
 FSi
 FSi
 FSi
 FSi
 FTQ
 Fin.
```

En Pascal, il existe une instruction très intéressante pour éviter de se farcir la tripotée de IF...THEN... C'est l'instruction CASE...OF. Voyons comment on pourrait traduire la série de IF...THEN de l'algo ci-dessus en Pascal (dans certaines versions du Pascal, l'instruction OTHERWISE doit être remplacée par ELSE).

```
...
 WHILE NOT sortir
 BEGIN
 WRITE('Quelle opération voulez-vous ? (+-*/) ') ;
 READLN(choix) ;
 CASE choix OF
 'Q','q': sortir := TRUE ;
 '+' : WRITELN(nombre1 + nombre2) ;
 '-' : WRITELN(nombre1 - nombre2) ;
 '*' : WRITELN(nombre1 * nombre2) ;
 '/' : WRITELN(nombre1 / nombre2) ;
 OTHERWISE
 WRITELN('Erreur dans votre choix') ;
 END ;
 END ;
 ...
```

Problème 6 :

Problème dit du palindrome. Un palindrome est une phrase qui peut être lue dans les deux sens. Par exemple : ELLE, LAVAL, ESOPE RESTE ICI ET SE REPOSE, etc... L'utilisateur entre une phrase terminée par un point et l'ordinateur lui dit s'il s'agit d'un palindrome. L'ordinateur ne doit pas tenir compte des espaces entre les mots pour la détermination du palindrome.

```
Lexique :
 phrase : chaîne de 50 caractères
 debut, fin : variables de type nombre entier qui vont servir de
 pointeur dans la chaine
 palindrome : variable de type booléen

 Debut
```

```
lire phrase
fin <-- longueur(phrase) *** fin := LENGTH(phrase) ; ***
debut <-- 1
palindrome <-- VRAI
TQ debut < fin ET palindrome
TQ phrase[debut] = ' '
debut <-- debut + 1
FTQ
TQ phrase[fin] = ' '
fin <-- fin - 1
FTQ
palindrome <-- phrase[debut] = phrase[fin]
debut <-- debut + 1
fin <-- fin - 1
FTQ
Si palindrome
Alors Afficher "C'est un palindrome"
Sinon Afficher "Ce n'est pas un palindrome"
FSi
Fin.
```

Il me semble que l'emploi d'un jeu d'essai est ici plus que jamais indispensable. Passez-y le temps qu'il faudra mais il est important que vous compreniez cet algo. La première boucle a pour but de tester s'il s'agit d'un palindrome. La première fois on entre obligatoirement dans la boucle puisque le booléen palindrome est mis à Vrai. Ensuite les deux petites boucles dans la grande ont pour but d'éliminer les blancs puisqu'on a dit qu'ils n'intervenaient pas dans la détermination du palindrome. Les "compteurs" debut et fin sont respectivement incrémentés et décrémentés de manière à se "déplacer" dans la chaine. Le booléen palindrome est ensuite testé pour vérifier si le caractère à la position "début" dans la chaîne "phrase" est égal au caractère à la position "fin" dans la même chaîne. Vous constatez qu'on peut donc "adresser" individuellement des éléments d'une chane, les modifier, les comparer, tout comme on le fera tout à l'heure avec les tableaux.

Problème 7 :

Ecrire un programme qui retourne la factorielle d'un nombre entré par l'utilisateur. Pour ceux qui ne le sauraient pas, ou qui l'auraient oublié, voici quelques exemples de calculs de factorielles :

- factorielle de 3 = 3 ! = 3 * 2 * 1 = 6

- factorielle de 4 = 4 ! = 4 * 3 * 2 * 1 = 24

Et voici l'algo... et l'occasion d'aborder l'utilisation des fonctions.

Programme principal

Lexique :

valeur : variable de type nombre réel

Début du programme principal

Lire valeur

Afficher valeur,' ! = ',factorielle(valeur)

Fin du programme principal

FUNCTION factorielle (nombre : REAL) : REAL

Lexique local :

resultat : variable locale de type nombre réel utilisée par la

fonction et par elle seule

Debut de la fonction

resultat <— 1

TQ nombre > 1

resultat <— resultat * nombre

nombre <— nombre - 1

FTQ

factorielle <— resultat

Fin de la fonction

Les puristes de l'algorithmique vont rêaler car j'ai pris quelques libertés quant à la déclaration de la fonction. En effet, je l'ai déclarée dans la syntaxe exacte du Pascal. Vous pouvez donc l'utiliser telle quelle en Pascal, si ce n'est que la traduction du reste de la fonction demeure à votre charge. L'usage de la fonction n'était pas ici obligatoire. J'aurais très bien pu intégrer le calcul de factorielle dans le programme principal. Vous pouvez d'ailleurs le faire si ça vous chante. Mais j'ai voulu vous présenter les fonctions, histoire d'avancer un peu et pour que ceux qui trouveraient que

je ne vais pas assez vite trouvent quand même leur compte. La fonction est un type de sous-programme. Si vous avez déjà utilisé des sous-programmes en Basic avec GOSUB et RETURN vous êtes déjà un petit peu familiarisé avec ces notions.

La fonction vue plus haut utilise trois notions :

- resultat : variable dite "locale" utilisée par la fonction pour effectuer les calculs. Cette variable, créée par la fonction lors de son appel, disparaêitra de la mémoire lors du retour au programme principal.

- nombre : cette variable fait office de paramètre "entrant". Lors de l'appel de la fonction par le programme principal, le contenu de la variable "valeur" est passé comme paramètre à la fonction "factorielle" et est placé dans la variable locale "nombre".

- factorielle : c'est le nom de la fonction (FUNCTION en Pascal) mais c'est aussi une variable locale à la fonction comme vous pouvez le constater dans l'algo, o| on affecte à la variable "factorielle" le résultat du calcul. Supprimez l'avant-dernière ligne de la fonction pour voir ce qui se passe. N'hésitez pas à modifier la fonction, quitte à provoquer des erreurs. C'est à mon avis une excellente façon d'apprendre.

Le sous-programme (ou PROCEDURE en Pascal) permet de créer des blocs d'instructions qui sont susceptibles d'être utilisés à plusieurs reprises par le programme principal ou par d'autres sous-programmes. La fonction de calcul de factorielle telle qu'elle est présentée ici peut être utilisée telle quelle dans n'importe quel autre programme par un simple Copier-Coller via le presse-papier. Une fonction comme celle-ci est en fait un type particulier de procédure. En algorithmique, les sous-programmes s'écrivent après le programme principal suivant en cela la structure de pensée rationnelle qui consiste à décomposer les différents problèmes en têaches élémentaires. En Pascal, par contre, il faudra écrire les fonctions et autres sous-programmes avant le programme principal. Lisez ce qui est dit sur la syntaxe des fonctions et procédures dans votre manuel. Il est intéressant de constater que la fonction peut utiliser ses propres variables, appelées variables locales, qui lors de l'exécution du programme seront créées en mémoire par l'appel à la fonction et disparaitront au moment du retour au programme principal permettant ainsi de "jouer" sur la taille de la mémoire. Imaginez un programme dont la taille en mémoire varierait au fur et à mesure de son exécution. C'est exactement ce qui se passe avec les fonctions et l'utilisation de variables locales. Les notions de variables locales et globales n'existent pas sur les vieux Basic o| toutes les variables sont globales. Par contre elles existent sur des Basic récents comme le GS.Basic. Il faut savoir aussi que les variables locales déclarées dans la fonction ne peuvent être utilisées par le programme principal. Par contre, les variables globales du programme principal peuvent être utilisées par les sous programmes et les fonctions. C'est un peu le principe du miroir sans teint qui permet de voir sans être vu. Ainsi la fonction peut "voir" et même modifier le contenu des variables utilisées par le programme principal. Cela nécessite d'être vigilant dans l'attribution des noms donnés aux variables, sinon bonjour les surprises.

Remarque : quand vous programmez en Pascal, vous utilisez sans le savoir un grand nombre de fonctions prédéfinies fournies avec le compilateur mais fonctionnant exactement comme celle que nous venons de voir. Par exemple l'instruction SQRT(arg) qui retourne la racine carrée de l'argument placé entre parenthèses est l'une de ces fonctions prédéfinies. L'instruction LENGTH utilisée dans le problème du palindrome est elle aussi une fonction. Voici quelques exemples d'utilisation :

...

valeur := 9 ; ** valeur est un REAL ou un INTEGER **

WRITELN(SQRT(valeur)) ;

toto := SQRT(valeur) ; ** toto est un REAL ou un INTEGER **

WRITELN(toto) ;

WRITELN('Dites ',SQRT(1089)) ;

READLN(phrase) ; ** phrase est une STRING **

WRITELN(LENGTH(phrase)) ;

golub := LENGTH(phrase) ; ** golub est un INTEGER **

phrase := phrase + 'coucou' ** concaténation de 2 chaînes **

WRITELN(phrase) ;

WRITELN(LENGTH(phrase)) ;

etc...

Faites des appels de ce type avec la fonction factorielle vue plus haut afin de vous familiariser avec cette notion très importante.

Chapitre 5 - Les fonctions et procédures

Problème 1 : (exemple de fonction)

Ecrire un programme qui retourne la factorielle d'un nombre entré par l'utilisateur. Pour ceux qui ne le sauraient pas, ou qui l'auraient oublié, voici quelques exemples de calculs de factorielles :

- factorielle de 3 = 3 ! = 3 * 2 * 1 = 6
- factorielle de 4 = 4 ! = 4 * 3 * 2 * 1 = 24

Et voici l'algo... et l'occasion d'aborder l'utilisation des fonctions.

```
Programme principal

 Lexique :
 valeur : variable de type nombre réel

 Début du programme principal
 Lire valeur
 Afficher valeur,'! = ',factorielle(valeur)
```

```
Fin du programme principal

-----------------------------------------------

FUNCTION factorielle (nombre : REAL) : REAL

Lexique local :
resultat : variable locale de type nombre réel utilisée par la
fonction et par elle seule

Debut de la fonction
resultat <-- 1
TQ nombre > 1
resultat <-- resultat * nombre
nombre <-- nombre - 1
FTQ
factorielle <-- resultat
Fin de la fonction
```

L'usage de la fonction n'était pas ici obligatoire. J'aurais très bien pu intégrer le calcul de factorielle dans le programme principal. Vous pouvez d'ailleurs le faire si ça vous chante. Mais j'ai voulu vous présenter les fonctions, histoire d'avancer un peu. La fonction est un type de sous-programme. Si vous avez déjà utilisé des sous-programmes en Basic avec GOSUB et RETURN vous êtes déjà un petit peu familiarisé avec ces notions.

La fonction vue plus haut utilise trois notions :

- resultat : variable dite "locale" utilisée par la fonction pour effectuer les calculs. Cette variable, créée par la fonction lors de son appel, disparaîtra de la mémoire lors du retour au programme principal.

- nombre : cette variable fait office de paramètre "entrant". Lors de l'appel de la fonction par le programme principal, le contenu de la variable "valeur" est passé comme paramètre à la fonction "factorielle" et est placé dans la variable locale "nombre".

- factorielle : c'est le nom de la fonction (FUNCTION en Pascal) mais c'est aussi une variable locale à la fonction comme vous pouvez le constater dans l'algo, o| on affecte à la variable "factorielle" le résultat du calcul. Supprimez l'avant-dernière ligne de la fonction pour voir ce qui se passe. N'hésitez pas à modifier la fonction, quitte à provoquer des erreurs. C'est à mon avis une excellente façon d'apprendre.

Le sous-programme (ou PROCEDURE en Pascal) permet de créer des blocs d'instructions qui sont susceptibles d'être utilisés à plusieurs reprises par le programme principal ou par d'autres sous-programmes. La fonction de calcul de factorielle telle qu'elle est présentée ici peut être utilisée telle quelle dans n'importe quel autre programme par un simple Copier-Coller via le presse-papier. Une fonction comme celle-ci est en fait un type particulier de procédure. En algorithmique, les sous-programmes s'écrivent après le programme principal suivant en cela la structure de pensée rationnelle qui consiste à décomposer les différents problèmes en tâches élémentaires. En Pascal, par contre, il faudra écrire les fonctions et autres sous-programmes avant le programme principal. Il est intéressant de constater que la fonction peut utiliser ses propres variables, appelées variables locales, qui lors de l'exécution du programme seront créées en mémoire par l'appel à la fonction et disparaitront au moment du retour au programme principal permettant ainsi de "jouer" sur la taille de la mémoire. Imaginez un programme dont la taille en mémoire varierait au fur et à mesure de son exécution. C'est exactement ce qui se passe avec les fonctions et l'utilisation de variables locales. Les notions de variables locales et globales n'existent pas dans certains langages (comme le

RPG) où toutes les variables sont globales. Il faut savoir aussi que les variables locales déclarées dans la fonction ne peuvent être utilisées par le programme principal. Par contre, les variables globales du programme principal peuvent être utilisées par les sous programmes et les fonctions. C'est un peu le principe du miroir sans teint qui permet de voir sans être vu. Ainsi la fonction peut "voir" et même modifier le contenu des variables utilisées par le programme principal. Cela nécessite d'être vigilant dans l'attribution des noms donnés aux variables, car certains compilateurs ne signalent pas les variables déclarées à la fois en global et en local.

Remarque : quand vous programmez en Pascal, vous utilisez sans le savoir un grand nombre de fonctions prédéfinies fournies avec le compilateur mais fonctionnant exactement comme celle que nous venons de voir. Par exemple l'instruction SQRT(arg) qui retourne la racine carrée de l'argument placé entre parenthèses est l'une de ces fonctions prédéfinies. L'instruction LENGTH utilisée dans le problème du palindrome est elle aussi une fonction. Voici quelques exemples d'utilisation :

```
...
 valeur := 9 ; ** valeur est un REAL ou un INTEGER **
 WRITELN(SQRT(valeur)) ;
 toto := SQRT(valeur) ; ** toto est un REAL ou un INTEGER **
 WRITELN(toto) ;
 WRITELN('Dites ',SQRT(1089)) ;
 READLN(phrase) ; ** phrase est une STRING **
 WRITELN(LENGTH(phrase)) ;
 golub := LENGTH(phrase) ; ** golub est un INTEGER **
 phrase := phrase + 'coucou' ** concaténation de 2 chaînes **
 WRITELN(phrase) ;
 WRITELN(LENGTH(phrase)) ;
 etc...
```

Faites des appels de ce type avec la fonction factorielle vue plus haut afin de vous familiariser avec cette notion très importante.

Problème 2 : (exemple de procédure)

Ecrire un programme qui lit les coefficients a, b et c d'une équation du second degré (aX2 + bX + C = 0) et la résout. Voilà là-encore une occasion de réviser vos connaissances de maths.

Attention, il faut prévoir le cas où le coefficient a est égal à 0. On en revient alors à une équation du premier degré du type : aX + b = 0. Si vous ne savez pas faire une résolution d'équation, c'est le moment ou jamais de l'apprendre en étudiant l'algo qui suit :

```
Programme principal

 Début
 Afficher 'Résolution d''une équation du type : aX2 + bX + c = 0'
 Lire coco,bobo,toto
 Faire equation(coco,bobo,toto)
 Fin.

 ----------------------------------------
```

```
Procedure equation(a,b,c : variables de type nombre réel)
*** a, b et c sont les paramètres entrants de la procédure equation ***

Lexique local
delta : variable locale de type REAL

Début de la procédure
Afficher 'Votre équation a donc la forme :'
Si a = 0
Alors Afficher b,'X + ',c,' = 0'
Si b <> 0
Alors Afficher 'La solution est ', -c/b
Sinon
Si c = 0
Alors Afficher 'La solution est R'
Sinon Afficher 'Pas de solution'
FSi
FSi
Sinon
Afficher a,'X2 + ',b,'X + ',c,' = 0'
Afficher 'Un instant, je calcule les solutions de X'
delta <-- b * b - 4 * a * c
Afficher 'Delta = ', delta
Si delta > 0
Alors
Afficher 'Votre équation a 2 solutions :'
Afficher 'X1 = ', (-b - SQRT(delta)) / (2 * a)
Afficher 'X2 = ', (-b + SQRT(delta)) / (2 * a)
Sinon
Si delta = 0
Alors
Afficher 'Votre équation n''a qu''une solution :'
Afficher ' X = ', -b / 2 * a
Sinon
Afficher 'Votre équation n''a pas de solution.'
FSi
FSi
FSi
Fin *** de la procédure ***
```

Il est nécessaire de comprendre qu'on n'aurait pas pu déclarer la procédure equation comme une fonction. En effet une fonction "retourne" obligatoirement un paramètre en sortie (comme dans factorielle ou SQRT) alors qu'une procédure peut faire beaucoup d'autres choses comme le montre la procédure equation.

Post-scriptum :

Cliquer ici pour les chapitres suivants.