Initiation à l'algorithmique 4 derniers 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 4 derniers chapitres :

Chapitre 6 - Les tableaux

Problème 1 :

On saisit 5 notes dans un tableau puis le programme affiche le contenu du tableau.

L'utilisation d'un tableau est ici intéressante car elle évite de devoir stocker les 5 notes dans 5 variables différentes du style note1, note2, note3, etc... ce qui entraînerait d'autres contraintes comme par exemple l'impossibilité de recourir à une boucle et un surplus de travail pour le programmeur. Si vous n'êtes pas convaincu, faites ce même programme sans recourir à un tableau et en utilisant 5 variables différentes, vous m'en direz des nouvelles.

```
Programme principal
 Lexique :
 notes : tableau(1 à 5) de nombres réels variables
 Début
 Faire saisie_notes
 Faire affiche_notes
 Fin.
 ------------------------------------
 Procédure saisie_notes
 Début du module
 i <-- 1
 TQ i <= 5
 Lire notes(i)
 i <-- i + 1
 FTQ
 Fin du module
 --------------------------------------
 Procédure affiche_notes
 Début du module
 i <-- 1
 TQ i <= 5
 Afficher notes(i)
 i <-- i + 1
 FTQ
 Fin du module
```

N.B. : Plusieurs points

- En Pascal, la déclaration du tableau se ferait ainsi : VAR notes : ARRAY[1..5] OF REAL ;
- Un FOR..TO..DO est particulièrement indiqué en lieu et place du Tant Que dans les 2 procédures.
- La gestion des tableaux diffère légèrement d'un langage à un autre. Dans certains langages, le premier poste d'un tableau est le poste zéro, dans d'autres langages, c'est le poste numéro un. Il faut être vigilant sur ce point en choisissant le nombre de postes du tableau.
- Les tableaux peuvent être déclarés comme globaux ou locaux, de la même façon que les variables. Mais certains langages, comme le RPG, ne connaissent que les variables et les tableaux globaux.
- Le tableau Notes est un tableau à une dimension. Nous verrons plus loin qu'il existe des tableaux à 2 dimensions ou plus. Le RPG ne gère que les tableaux à une dimension, Adélia sait gérer des tableaux à 2 dimensions, d'autres langages tels que le C ou le Pascal savent gérer des tableaux de plus de 2 dimensions.
- Quand vous serez familiarisé avec les tableaux, je vous recommande de lire l'article concernant l'optimisation de recherche dans un tableau.

Je parie que ce qui vous interpelle le plus dans l'algo ci-dessus, c'est que l'on puisse écrire "Lire notes(i)". Eh bien oui, on peut affecter ainsi une valeur à une "case" d'un tableau sans autre forme de procès comme on pourrait le faire avec les chaînes. Ecrivez-vous de petits programmes utilisant des tableaux de petite dimension. Utilisez de petits exercices du même type que ceux que je vous ai présentés pour l'utilisation de la chaîne phrase. Vous constaterez par exemple qu'on ne peut visualiser un tableau en totalité comme on le ferait pour une chaîne avec WRITELN(notes) par exemple. Il faut pour cela recourir à une boucle comme dans le module affiche_notes.

Il ne vous est pas interdit de déclarer un tableau comme suit :

```
Produit : Tableau[15 à 255] de variables de type nombre entier
```

En Pascal, cela donnerait :

Produit : ARRAY[15..255] OF INTEGER ;

Dans certaines versions du Basic, il faudrait écrire :

```
DIM Produit%(255)
```

Donc en Pascal, le tableau contiendra 241 éléments numérotés de 15 à 255 tandis qu'en Basic, ce même tableau contiendra 256 éléments numérotés de 0 à 255 (en Basic on ne peut pas faire autrement). On pourra néanmoins ne travailler que sur les éléments numéro 15 à 255 si ça nous chante, mais ceci au prix d'une perte de place mémoire. En RPG, et en Adélia, on retrouve la même limitation qu'en Basic.

Il vous est aussi possible d'écrire :

```
Nom_jour : Tableau(Lundi à Dimanche) de chaînes
```

ou encore :

```
Montant_ventes : Tableau(mini à maxi) de réels
```

N'oubliez pas que l'on peut faire des tableaux de pratiquement n'importe quoi (CHAR, STRING, REAL, BOOLEAN, etc...), alors essayez, explorez, bidouillez, faites des essais, c'est le meilleur moyen pour comprendre.

Un petit exercice : écrivez un troisième module qui lira le tableau notes et affichera la note la plus haute ainsi que sa "position" dans le tableau. Ca ne vous rappelle rien ? Quand vous serez suffisamment aguerri, passez à l'étude du...

Problème 2 :

On saisit en vrac les ventes du mois à partir de bordereaux qui sont dans le désordre. L'utilisateur entre le numéro de mois indiqué sur le bordereau, puis le montant des ventes indiqué sur ce même bordereau. S'il n'a plus de bordereau à saisir, il entre un numéro de mois égal à 0 et le programme édite pour chaque mois : son nom, le nombre de ventes et le montant des ventes.

Pour les masochistes qui ne seraient toujours pas convaincus de l'utilité des tableaux, je conseille de faire ce même programme sans recourir à des tableaux, c'est à dire en définissant 36 variables différentes, puisqu'on a besoin ici de 3 tableaux de 12 éléments chacun (1 pour stocker le montant des ventes de chaque mois, 1 pour stocker le nombre de ventes de chaque mois, et 1 pour stocker le nom de chaque mois). Si vous y arrivez... Bravo !!! (mais un bon conseil, laissez tomber la programmation car à mon avis c'est sans espoir).

Blague à part, avant d'attaquer l'étude de l'algo proprement dit, je voudrais vous indiquer la manière de stocker

en mémoire le nom des mois. Pour ce faire j'utilise un tableau de chaînes stocké en constante, ce qui évite à l'utilisateur de devoir entrer le nom des mois à chaque fois qu'il lance le programme. Voici comment écrire cela en Pascal :

```
CONST

tabnommois : ARRAY(1..12) OF STRING(9) = ('Janvier','Février','Mars',
 'Avril','Mai','Juin','Juillet','Août','Septembre',
 'Octobre','Novembre','Décembre') ;
```

J'ai choisi ici une écriture très modulaire car chaque sous-programme peut ainsi être étudié indépendamment. Imprimez cette partie et amusez-vous à "désosser" chaque module pour bien comprendre leur fonctionnement. Regardez bien qui fait quoi et qui appelle qui. Repérez les variables locales et globales. Faites-vous des jeux d'essai. N'oubliez pas qu'en Pascal, vous devrez écrire tous les modules avant le programme principal... j'y reviendrai à la fin de cet article. On pourrait encore optimiser la solution proposée ci-dessous mais je ne tiens pas à compliquer outre mesure. Revenons à nos moutons :

```
Programme principal

 Lexique des Variables globales :
 nombre_ventes : TABLEAU(1 à 12) de nombres entiers
 montant_ventes : TABLEAU(1 à 12) de nombres réels
 num_mois : variable de type nombre entier
 ventes_lues : variable de type nombre réel

 Début
 Faire init_tableau
 Faire saisie_ventes
 Faire edition_ventes
 Fin.

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

 Procédure init_tableau
 *** On commence par initialiser les 2 tableaux à 0 ***
 Début ** du module **
 num_mois <-- 1
 TQ num_mois <= 12
 nombre_ventes(num_mois) <-- 0
 montant_ventes(num_mois) <-- 0.
 num_mois <-- num_mois + 1
 FTQ ** Initialisation terminée **
 Fin ** du module **
```

Explication du module init_tableau : l'initialisation des tableaux nombre_ventes et montant_ventes est indispensable car on fait ici des saisies en vrac et on n'est pas sûr qu'en fin de saisie, tous les mois aient forcément des ventes. C'est une précaution indispensable qui permet d'éviter que des valeurs parasites ne viennent interférer dans les calculs. Ca vous parait bizarre ? Si vous lancez un programme plusieurs fois, votre programme risque de se retrouver avec le contenu des variables utilisées lors de sa précédente exécution. Bonjour les bugs ! A noter aussi qu'il vaut mieux initialiser un tableau de réels avec "0." que "0" tout court, sinon vous risquez de chercher un moment pourquoi votre programme "débloque" (et là le TML/Pascal ne fait pas exception) car le compilateur "attend" un réel et vous lui donnez quelque chose qu'il prend pour un entier. Je sais c'est idiot mais c'est comme ça. Ne demandez pas à votre compilateur préféré qu'il fasse preuve

d'intelligence. Pour en revenir à notre module, vous remarquez qu'on utilise une seule boucle pour initialiser les 2 tableaux en même temps. L'initialisation d'un tableau (surtout de cette taille) nécessite très peu de temps machine comme vous pourrez le constater, alors autant la faire.

En fait, la procédure Init_tableau pourrait s'écrire aussi comme ceci :

```
Procédure init_tableau
 *** On commence par initialiser les 2 tableaux à 0 ***
 Début ** du module **
 nombre_ventes <-- 0
 montant_ventes <-- 0.
 Fin ** du module **
```

La plupart des langages acceptent qu'on initialise le tableau globalement sans passer par une boucle d'initialisation de chaque poste.

```
Module saisie_ventes
 *** On saisit les ventes de chaque mois en vrac ***
 Début du module
 Lire num_mois
 TQ num_mois <> 0
 Lire ventes_lues
 nombre_ventes(num_mois) <-- nombre_ventes(num_mois) + 1
 montant_ventes(num_mois) <-- montant_ventes(num_mois) + ventes_lues
 Lire num_mois
 FTQ
 Fin du module
```

L'utilisateur entre le numéro du mois. Si le numéro est différent de 0, on entre dans la boucle. L'utilisateur saisit alors le montant des ventes de ce mois, montant qui est cumulé à celui déjà existant dans la "case" num_mois du tableau montant_ventes. De même on incrémente de 1 le contenu de la "case" num_mois dans le tableau nombre_ventes. Puis on entre un nouveau numéro de mois et rebelote.

```
Module edition_ventes
 *** édition des ventes des mois 1 à 12 ***

 Lexique:
 Constante locale :
 tabnommois : TABLEAU(1 à 12) de chaines de caractères

 Début du module
 Afficher ' Mois Nbre de ventes Total des ventes'
 num_mois <-- 1
 TQ num_mois < 13
 Afficher tabnommois(num_mois), nombre_ventes(num_mois),
 montant_ventes(num_mois)
 num_mois <-- num_mois + 1
 FTQ
```

```
Fin du module.
```

La principale fonction de ce module est de lire les tableaux tabnommois, nombre_ventes et montant_ventes simultanément et entièrement (des "cases" 1 à 12 pour être précis) afin d'afficher leur contenu. En Pascal on pourra écrire la boucle ainsi : FOR num_mois := 1 TO 12 DO...

Probléme 3 :

Nous allons maintenant reprendre le problème 3 de ce chapitre pour l'améliorer. En plus de la saisie des ventes en vrac, le programme permettra l'édition au choix :

- de tous les mois,

- d'un mois (le choix du mois est laissé à l'initiative de l'utilisateur),

- d'une période (de juillet à novembre (mois 7 à 11), par exemple, mais là encore, c'est l'utilisateur qui décide).

Pour ce faire il suffira de modifier uniquement le module edition_ventes en lui ajoutant un menu et quelques petits aménagements.

```
Module edition_ventes

 Variables locales :
 sortir : booléen (tiens, ça faisait longtemps !)
 choix : CHAR (j'ai la flemme, de toute façon vous avez compris)
 mois_num1, mois_num2 : INTEGER

 Début
 sortir <-- FAUX
 TQ non sortir
 Afficher '1 - Edition de tous les mois..'
 Afficher '2 - Edition d''un mois..'
 Afficher '3 - Edition d''une période..'
 Afficher '9 - Fin de session.'
 Afficher 'Entrez votre choix : '
 Lire choix
 Si choix = '9'
 Alors sortir <-- VRAI
 Sinon
 Si choix = '1'
 Alors
 Faire periode(1,12)
 Sinon
 Si choix = '2'
 Alors
 Afficher 'Entrez le numéro du mois'
 Lire mois_num1
 Faire periode(mois_num1,mois_num1)
 Sinon
 Si choix = '3'
```

```
 Alors
 Afficher 'Entrez la période désirée'
 Lire mois_num1, mois_num2
 Faire periode(mois_num1,mois_num2)
 FSi
 FSi
 FSi
 FSi
 FTQ
 Fin du module
```

Voilà un programme de menu classique. Observez de quelle façon est appelé le module periode. C'est important. N'hésitez pas à vous servir de l'instruction CASE...OF... pour la transcription en Pascal.

```
Procedure periode(debut,fin : INTEGER)
 *** debut et fin sont les paramètres entrants ***

 Lexique:
 Constante locale :
 tabnommois : TABLEAU(1 à 12) de chaines de 9 caractères

 Début de la procédure
 Si debut > fin
 Alors
 Afficher 'Mauvais paramètres, recommencez'
 Sinon
 Afficher ' Mois Nbre de ventes Total des ventes'
 num_mois <-- debut
 TQ num_mois <= fin
 Afficher tabnommois(num_mois), nombre_ventes(num_mois),
 montant_ventes(num_mois)
 num_mois <-- num_mois + 1
 FTQ
 FSi
 Fin du module.
```

Comparez ce module avec la première version du module edition_ventes.

Peut être n'est-il pas inutile de rappeler la logique de notre programme de traitement des ventes :

```
Programme principal
 !
 !---- Module init_tableau
 !---- Module saisie_ventes
 !---- Module edition_ventes
 !
 !---- Module periode
```

Pour la transcription en Pascal, vous devrez écrire les différents modules dans l'ordre suivant :

1. Module init_tableau
2. Module saisie_ventes
3. Module periode
4. Module edition_ventes
5. Programme principal

L'ordre dans lequel vous mettrez les modules init_tableau, saisie_ventes et edition_ventes importe peu à condition de les déclarer avant le programme principal. En revanche, la procédure periode (qui est le module appelé) doit obligatoirement se trouver avant le module appelant (en l'occurrence edition_ventes). C'est une règle incontournable.

Contraintes des vieux langages :

Un ami avec qui je discutais d'algo me demanda comment traduire en Basic, en "collant" le plus près possible à l'algorithme, des instructions telles que TantQue...FindeTantQue ou Si...Alors...Sinon...FindeSi. Il n'est peut être pas inutile d'y revenir, il serait dommage que les utilisateurs du Basic se sentent "largués" alors que dans la plupart des cas, on peut traduire les algos vus jusqu'ici dans ce bon vieux Basic Applesoft sans problèmes. Le prix d'un compilateur Pascal est souvent dissuasif alors que l'Applesoft est livré gratuitement avec tous les Apple II.

```
Si... Alors... Sinon... FSi :

 Début
 Lire note
 Si note < 8
 Alors
 Afficher 'candidat éliminé'
 Sinon
 Si note < 10
 Alors Afficher 'candidat admis à l'épreuve de rattrapage'
 Sinon Afficher 'candidat reçu'
 FinSi
 FinSi
 Fin.
```

En Basic Applesoft, cela donnerait :

```
10 INPUT NOTE
 20 IF NOTE < 8 THEN PRINT "CANDIDAT ELIMINE":GOTO 50
 30 IF NOTE < 10 THEN PRINT "CANDIDAT ADMIS A L'EPREUVE DE RATTRAPAGE":GOTO 50
 40 PRINT "CANDIDAT RECU"
 50 END
```

C'est ça que je voulais dire en parlant de "jongler avec les GOTO du Basic" dans la première partie de ce cours :

```
TQ...FTQ :
 i <-- 0
 TQ i <= 7
 Afficher i
 i <-- i + 1
```

```
 FTQ
```

Pourrait se traduire par :

```
10 FOR I=0 TO 7
 20 PRINT I
 30 NEXT I
```

Ou encore :

```
10 I=0
 20 IF I>7 THEN 60
 30 PRINT I
 40 I=I+1
 50 GOTO 20
 60 <Suite du programme>
```

Remarquez l'inversion de la condition de sortie de la boucle en ligne 20.

En résumé l'algo qui suit (vous le reconnaitrez sûrement) :

```
Début
 gagne <-- nombre pris au hasard entre 1 et 100
 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.
```

Pourra se traduire ainsi :

```
 10 GAGNE = INT(RND(1)*100)
 20 INPUT NBRELU
 30 N=1
 40 IF NBRELU = GAGNE THEN 100
 50 IF NBRELU > GAGNE THEN PRINT "TROP GRAND !":GOTO 70
 60 PRINT "TROP PETIT !"
 70 N=N+1
 80 INPUT NBRELU
 90 GOTO 40
 100 PRINT "VOUS AVEZ TROUVE EN ";N;" COUP(S)."
```

Autant en Basic qu'en Applesoft, il faut veiller à ne pas utiliser de mots réservés comme variables, sous peine de réactions inattendues du programme dont la plus courante est le SYNTAX ERROR.

Sous-programmes : si vous espériez adapter les modules avec passage de paramètre (procédures et fonctions) que nous avons vues dans le cours précédent, vous en serez pour vos frais (à moins que vous ne travailliez

dans des versions modernes du Basic, ce que je vous souhaite). Mais ce n'est pas grave, on peut facilement s'en passer moyennant quelques aménagements. De toute façon, les notions de constantes, de passage de paramètre et de variables locales n'existent pas sur les vieux Basic comme l'Applesoft (de même qu'en RPG). Sur ces vieux Basic, toutes les variables sont globales. Il suffira de garder ces contraintes à l'esprit en écrivant l'algo pour éviter les mauvaises surprises lors de la traduction en Basic. Voilà schématiquement comment on pourrait traduire le problème 6 du cours précédent :

```
10 REM <Titre, Auteur, Date de création>
 30 GOSUB 1000
 40 GOSUB 2000
 50 GOSUB 3000
 60 END
 1000 REM MODULE INIT-TABLEAU
 <contenu du sous-programme>
 1200 RETURN
 2000 REM MODULE SAISIE-VENTES
 <contenu du sous-programme>
 2030 RETURN
 3000 REM MODULE EDITION-VENTES
 <contenu du sous-programme>
 3050 RETURN
```

Un bon exercice : reprenez le problème 3 de ce chapitre, réécrivez l'algo de la procédure PERIODE pour en faire un module sans passage de paramètre, puis modifiez le module EDITION-VENTES en conséquence et mettez tout ça en Basic à la suite du programme ci-dessus (sans oublier de compléter les modules INIT-TABLEAU et SAISIE-VENTES qui en ont bien besoin).

En tout état de cause, n'hésitez pas à viser large en ce qui concerne les numérotations de lignes. Ainsi si vous voulez opérer des modifications ou des ajouts de lignes, vous n'aurez pas trop de problèmes. Vous remarquerez que si vous avez bien travaillé votre algo (jeux d'essai et tout le tra-la-la), son écriture en Basic avec la contrainte des numéros de lignes ne relève plus que de l'exercice de style. Le programme "plat de spaghetti" n'est plus alors qu'un lointain souvenir (ou presque, tout est relatif). N'hésitez pas à aérer vos programmes par des commentaires, voire des lignes de REM "vides", ceci afin de faciliter la relecture et le débogage, de toute façon inévitable. Une solution agréable à laquelle ne pensent généralement pas les débutants consiste à taper son programme sur un traitement de texte ou un éditeur quelconque, à le sauvegarder sous forme de fichier texte et à le récupérer en tapant sous Applesoft :

```
EXEC <nom du fichier>
```

Là où les choses se corsent, c'est lorsque vous voulez reprendre un fichier Basic sous un traitement de texte. Je vous conseille de vous reporter à l'article de Dimitri Geystor dans la revue Pom's n°38, qui apporte une solution intéressante au problème.Passons maintenant aux booléens :

La notion de booléen existe en Applesoft même s'il n'existe pas de variable booléenne proprement dite (la variable gagne ci-dessous est un booléen) :

```
Lire nbrelu
 gagne <-- nbrelu > 45
```

Pourra se traduire par :

```
40 INPUT NBRELU
 50 GAGNE = NBRELU > 45
```

Notez qu'il est impossible d'afficher le contenu d'une variable booléenne en algo. En Basic (avec un PRINT) on peut le faire car on utilise une variable numérique.

Si la condition est vraie la variable booléenne "gagne" recevra la valeur "vraie" en algorithmique et la variable numérique "gagne" recevra le nombre 1 en Basic. Dans le cas contraire la variable booléenne recevra "faux" et la variable numérique recevra le nombre 0. On utilisera une variable numérique à la place d'une variable booléenne mais le résultat est le même et c'est ce qui compte finalement. Pour vous en convaincre, faites des essais du type :

```
? 15<=13
 ? 5<9
 A=3>9
 ? A
```

Exemple

```
Si Non gagne
 Alors Afficher 'Vous avez perdu !'
 Sinon Afficher 'Vous avez gagné !'
 FSi
70 IF GAGNE=0 THEN PRINT "VOUS AVEZ PERDU !":GOTO 90
 80 PRINT "VOUS AVEZ GAGNE !"
 90 <suite du programme>
```

Problème 4 :

Constitution d'un annuaire et recherche d'une personne et de son numéro de téléphone dans l'annuaire.

Commençons par constituer 2 tableaux, l'un servant à stocker le nom des personnes, l'autre servant à stocker les numéros de téléphone correspondant. Deux tableaux de chaines me semblent ici particulièrement indiqués, tableaux que nous appellerons nom et telephon.

```
Programme principal

 Lexique des variables et constantes globales
 max : constante qui indiquera la taille maximum des tableaux utilisés.
 En la modifiant, on modifiera automatiquement la taille des
 tableaux. Pour commencer, mettons la à 100.
 nom : tableau(1 à max) de chaines de 15 caractères
 telephon : tableau(1 à max) de chaines de 11 caractères.
 dernier : variable de type nombre entier qui doit être inférieure à
 max, ce qui nous permettra, nous le verrons plus tard, de la
 faire évoluer en insérant ou en supprimant des personnes dans
 le fichier.
 i : variable de type nombre entier qui sert de compteur.

 Début (du programme principal)
 Faire saisie_annuaire
 Faire recherche
```

```
Fin (du programme principal)

Module saisie_annuaire
Début
i <-- 1
dernier <-- max
TQ dernier > max - 10
Afficher message d'erreur
Lire dernier
FTQ
TQ i <= dernier
Lire nom(i), telephon(i)
i <-- i + 1
FTQ
Fin
```

Vous commencez par indiquer le nombre de personnes que vous désirez saisir. Le programme effectue un contrôle sur dernier en le comparant à max-10 afin de s'assurer que vous lui laissez une marge suffisante pour les éventuelles insertions de nouvelles personnes (nous verrons ça plus tard, mais autant être prévoyant). Puis vous passez à la saisie, pour chaque personne de son nom et de son numéro de téléphone. Vous remarquerez que les traitements, notamment la recherche, ne se feront jamais de 1 à max mais de 1 à dernier. Il est en effet inutile de travailler sur la partie du tableau comprise entre dernier et max puisqu'on n'a rien saisi dedans.

```
Module recherche

Lexique local :
nom_corresp : variable de type chaine de caractères
trouve : variable de type booléen

Début
Lire nom_corresp
i <-- 1
trouve <-- faux
TQ i <= dernier Et Non trouve
trouve <-- nom(i) = nom_corresp
i <-- i + 1
FTQ
Si non trouve
Alors Afficher 'Ce correspondant n''existe pas dans votre tableau'
Sinon Afficher nom(i-1), telephon(i-1)
FSi
Fin
```

Un petit coup main aux basiqueurs :

```
2000 REM ** MODULE RECHERCHE **
 2010 INPUT CORRESP
 2020 I = 1
 2030 TROUVE = 0
 2040 IF (I > DERNIER) OR (TROUVE = 1) THEN 2080
 2050 TROUVE = NOM(I) = CORRESP
 2060 I = I + 1
 2070 GOTO 2040
```

```
2080 IF... (pour le reste débrouillez-vous)
```

Observez l'inversion de la condition en ligne 2040 par rapport au TQ.

Vous remarquez que l'on fait ici une recherche par lecture séquentielle du tableau NOM. Il faudra veiller à l'orthographe du nom recherché, sinon le programme vous dira qu'il n'existe pas même si vous savez qu'il s'y trouve. De plus on ne traite pas ici les cas d'homonymie mais c'est un détail (qu'il pourra être intéressant de régler, par exemple en modifiant le programme pour qu'il affiche toutes les personnes ayant le même nom).

Une recherche séquentielle, c'est bien joli sur un tableau peu important, mais imaginez ce que ça pourrait donner sur un tableau de 1000 éléments ou plus. Vous aurez alors tout le loisir d'aller tondre votre pelouse en attendant que votre programme trouve la personne que vous lui indiqué (en supposant qu'il la trouve). Bien sûr j'exagère mais nous allons quand même étudier un outil très puissant que l'on appelle recherche dichotomique et qui permet de trouver un élément dans un tableau en un temps record.

La recherche dichotomique, qu'est-ce au juste ? Pour l'illustrer, prenons l'exemple du programme de devinette d'un nombre. Supposons que le nombre à trouver soit compris entre 1 et 100 et soit en l'occurrence 39. Vous n'en savez rien et vous tenez à trouver ce nombre avec le minimum de coups. Si vous êtes quelqu'un de très rationnel qui ne laisse rien au hasard, vous allez commencer par entrer 50. L'ordinateur vous indique "Trop Grand", vous savez donc que votre nombre est compris entre 1 et 49. Coupez la poire en deux et entrez 25. L'ordinateur vous indique "Trop petit" et vous en concluez que le nombre est compris entre 26 et 49. (26 + 49) / 2 ça fait 37,5 alors vous indiquez 37 et l'ordinateur vous indique encore "Trop petit". Vous savez donc que votre nombre est compris entre 38 et 49... Il est inutile que je continue, vous avez compris - en tout cas, je l'espère. Tel monsieur Jourdain qui faisait de la prose sans le savoir, vous venez de faire en ma compagnie la recherche dichotomique d'un nombre, pas dans un tableau certes, mais le principe est le même. On peut estimer que vous aurez trouvé le nombre 39 en moins d'une dizaine de coups, alors que vous aviez contre vous 100 possibilités : sans commentaires.

La recherche dichotomique n'est possible que sur des tableaux triés, par exemple par ordre croissant pour les tableaux de nombres, par ordre alphabétique pour les tableaux de caractères ou de chaînes. Cela implique que vous ayez pris la peine de trier vos correspondants par ordre alphabétique avant de les saisir avec le module saisie_annuaire. Les bouquins d'algorithmique sont pleins d'algo de tri, c'est un sujet complexe et passionnant. Je vous rassure, on n'est pas obligé de devenir un pro du tri pour concevoir des programmes, car de nombreux environnements de développement regorgent de fonctions de tri et de recherche (sur tableau et fichier) qui sont destinées à simplifier la vie du programmeur et lui évitent de réinventer la roue.

De plus, la recherche dicho n'a d'intérêt, en termes de temps machine, que sur des tableaux dépassant une vingtaine d'éléments. En deça de cette limite la recherche séquentielle est acceptable d'autant qu'elle ne nécessite pas que le tableau soit trié. Il pourra être amusant de combiner dans un même programme l'algo de recherche séquentielle et l'algo de recherche dicho. Par exemple, si le nombre d'éléments (contenu ici dans la variable DERNIER) est inférieur à une limite que vous aurez déterminée, le programme lancera la recherche séquentielle, sinon ce sera la recherche dicho : une façon comme une autre d'optimiser les performances d'un programme.

Cherchez un moment comment écrire l'algo de la recherche dichotomique dans le tableau NOM puis lisez l'algo qui suit :

```
Module recherche

 Lexique local :
```

```
debut, fin, milieu : variables de type nombre entier
trouve : booléen
nom_corresp : variable de type chaine de caractères

Début
Lire nom_corresp
debut <-- 1
fin <-- dernier
trouve <-- faux
TQ debut <= fin Et Non trouve
milieu <-- (debut + fin) DIV 2
Si nom_corresp < nom(milieu)
Alors
fin <-- milieu - 1
Sinon
Si nom_corresp > nom(milieu)
Alors debut <-- milieu + 1
Sinon trouve <-- vrai
FSi
FSi
FTQ
Si Non trouve
Alors Afficher 'Ce correspondant n''existe pas dans votre tableau'
Sinon Afficher nom(milieu), telephon(milieu)
FSi
Fin
```

Voilà, je ne crois pas avoir fait d'erreur. Faites-vous un jeu d'essai sur les variables : debut, fin, milieu, trouve, nom. Ne vous laissez pas impressionner et potassez cet algo jusqu'à l'avoir bien maîtrisé. J'entends par "maîtriser" le fait d'être capable de l'appliquer sur toutes sortes de tableaux. Une fois qu'on a compris le principe, ça ne doit plus poser de problème.

Il faut noter que la recherche dicho permet de trouver la position dans le tableau où devrait se trouver la personne recherchée, ce que ne permet pas directement la recherche séquentielle. Ainsi, nous saurons très exactement où insérer dans le tableau une personne qui n'y est pas encore. Essayez de trouver laquelle des variable "debut", "fin" ou "milieu" permet de trouver cette position.

N.B. : La variable qui permet de trouver la position de l'élément que l'on recherche est la variable "debut". Ca c'est pour le petit à lunette, au premier rang, qui a l'air de suivre.

Chapitre 7 - La récursivité

Vous avez tous vu l'exemple de la caméra vidéo branchée sur un téléviseur et dont l'objectif est pointé sur l'écran du téléviseur. Vous voyez alors sur l'écran du téléviseur ce même téléviseur qui se répète à l'infini en lui-même.

Prenons maintenant la définition donnée dans le Dictionnaire de l'Informatique des éditions Larousse : "Technique de réduction de la complexité d'un problème permettant d'obtenir des programmes fiables et

efficaces... L'utilisation de la récursivité donne un procédé d'analyse très puissant pour beaucoup de problèmes. Cependant, certains langages de programmation n'offrent pas la possibilité d'écrire des programmes récursifs. Des techniques de transformation de programmes permettent de se ramener à un traitement séquentiel, par utilisation de piles permettant de gérer les différents niveaux de paramètres."

Retenons déjà que les langages non procéduraux comme le bon vieux Basic Applesoft ne peuvent exploiter cette technique. Le comble, c'est que tous les langages procéduraux ne permettent pas au programmeur d'exploiter la notion de pile (c'est le cas du RPG). Tant pis, ce n'est pas ça qui va me priver du plaisir de vous présenter la récursivité. En tout cas, voilà une notion que les programmeurs en Forth ne manqueront pas d'exploiter, la gestion de la pile étant un des points forts de ce langage. Qu'est-ce qu'une pile (stack en anglais) ? Le dico de l'info cité plus haut dit ceci : "File d'attente gérée selon la méthode du dernier entré - premier sorti. (L'accès aux informations se fait dans l'ordre inverse où celles-ci ont été rangées)."

Retenons aussi qu'un programme récursif est un programme qui s'appelle lui-même mais et c'est le plus important - a une condition de terminaison. Supposons donc que par erreur vous ayez écrit dans le module edition_ventes la phrase suivante : Faire edition_ventes. On ne pourra pas parler ici de traitement récursif mais plutôt d'un gros bug qui risque de vous planter magistralement, la procédure s'appelant elle-même jusqu'à saturation de la pile.

Trêve de bavardages, prenons un cas concret qui, je l'espère, vous permettra de mieux comprendre ce qu'est la récursivité. Vous vous souvenez de la fonction factorielle vue la dernière fois ? Eh bien nous allons la reprendre et la modifier pour la rendre récursive :

```
Programme principal

 Lexique des variables globales :
 nombre, result : variables de type REAL

 Début
 Lire nombre
 result <-- factorielle(nombre)
 Afficher result
 Fin
```

N.B. : ne vous laissez pas surprendre par les numéros de lignes dans la fonction factorielle ci-dessous, je les ai ajoutés pour m'aider dans les explications qui vont suivre, mais ils n'ont aucune incidence sur l'algo lui-même.

```
Fonction factorielle (n : réel) : réel
 ** "n" est le paramètre entrant, "factorielle" le paramètre sortant **
 10 Début
 20   Si n = 1
 30     Alors factorielle <-- 1
 40     Sinon factorielle <-- factorielle(n - 1) * n
 50   FSi
 60 Fin
```

Vous trouvez cet algo incompréhensible ? Alors pensez à moi qui vais devoir vous l'expliquer. La récurvisité, c'est l'exemple type de notion qui est plus difficile à expliquer qu'à comprendre. En tout cas, j'espère que vous maitrisez bien les fonctions maintenant, sinon on est mal parti.

Le programme principal ne présente pas de difficulté : Supposons que le nombre envoyé à la fonction soit 3.

Factorielle de 3 ça fait 6. Voyons si la fonction l'entend de cette oreille.

3 est donc passé en paramètre d'entrée à la variable n. La condition n'étant pas réalisée en ligne 20 on passe à la ligne 40 :

```
factorielle <-- factorielle(3-1) * 3
```

L'ordinateur commence par calculer factorielle(n-1) et pour ce faire rappelle la fonction factorielle en lui passant (n-1) - c'est à dire 2 - comme paramètre. Imaginez que vous venez de poser la première assiette d'une pile nommée factorielle. On "repasse" en ligne 20 et la condition n'étant toujours pas réalisée, on repasse en ligne 40 :

```
factorielle <-- factorielle(2-1) * 2
```

De nouveau, le programme lance la fonction factorielle avec 1 comme paramètre. On pose la seconde assiette factorielle sur la première. Cette fois la condition en ligne 20 est bien réalisée donc on passe à la ligne 30 :

```
factorielle <-- 1
```

Cette ligne apparemment anodine représente la condition de terminaison. En effet, cette fois la fonction n'est pas appelée une nouvelle fois par elle-même. On passe donc en ligne 50, puis 60 et on "quitte" la fonction pour revenir à la fonction immédiatement supérieure, c'est à dire qu'on envoie valser l'assiette numéro 2 pour revenir sur l'assiette numéro 1 avec en paramètre de sortie le contenu de la variable factorielle qui est 1. Nous avions "quitté" l'assiette 1 en ligne 40, nous revenons donc en ligne 40 pour terminer le calcul que nous y avions commencé, c'est à dire :

```
factorielle <-- factorielle(2-1) * 2
```

Puisque la factorielle de 1 a été calculée et est égale à 1, la ligne ci-dessus équivaut à la ligne ci-dessous :

```
factorielle <-- 1 * 2
```

Ainsi le calcul en ligne 40 est terminé et nous passons en ligne 50 puis 60 pour "quitter" la fonction factorielle avec comme paramètre de sortie le résultat de l'opération qui est 2, et revenir à la fonction de rang immédiatement supérieur. En d'autres termes, on éjecte l'assiette numéro 1 et on vient se placer dans l'assiette numéro 0, c'est à dire la fonction factorielle d'origine, celle qui a été appelée par le programme principal. Nous étions "sortis" de cette dernière à la ligne 40, nous y revenons donc pour achever le calcul :

```
factorielle <-- factorielle(3-1) * 3
```

qui équivaut à ce que j'écris ci-dessous :

```
factorielle <-- 2 * 3
```

Ainsi nous quittons définitivement la fonction factorielle avec 6 en paramètre de sortie. Imaginez que vous venez

de briser l'assiette 0 d'un coup de masse rageur. La récurvisité ça défoule. En fin de compte, nous revenons au programme principal avec le résultat de la factorielle de 3 qui est bien 6.

Lisez et relisez ce que je viens d'écrire, aidez-vous d'un tableau de ce type :

```
Numéro d'assiette ! Paramètre d'entrée ! Paramètre de sortie
 -----------------!-------------------!--------------------
 0          !          3         !         6
 1          !          2         !         2
 2          !          1         !         1
```

Faites-vous des jeux d'essai, des schémas, amusez-vous à faire tourner ce programme "à la main" (comme je viens de le faire) avec la factorielle de 4, de 5 ou de ce que vous voulez.

Il y aurait encore beaucoup à dire sur la récurvisité. On peut par exemple écrire un programme qui résolve le problème des Tours de Hanoï par un traitement récursif. Il existe un algo de tri de tableau extrêmement rapide sur des tableaux de grande taille et qui utilise la récursivité : c'est le QuickSort (tri rapide). Bref, une multitude d'applications en perspective.

Chapitre 8 - Les tableaux à 2 dimensions

Nous allons étudier les tableaux à 2 dimensions sur de petites routines de traitement d'image. Bidouiller des images en 2 dimensions telles que des dessins ou photos est à mon avis un très bon moyen de se familiariser avec la programmation sur des tableaux à 2 dimensions car les mécanismes (que ce soient des pixels ou des montants de ventes) sont les mêmes.

Problème 1 : essayez donc de deviner ce que fait ce programme

```
Programme principal

 Constantes globales :
 XM = 319
 YM = 199

 Variables globales :
 Ecran : Tableau(0 à XM, 0 à YM) de nombres entiers
 X, Y : variable de type nombre entier

 Début
 Faire Initialisation écran graphique
 Faire HASARD
 Faire RECOPIE
 Faire fermeture des Outils
 Fin

 Module HASARD
 Début
 Pour X allant de 0 à XM
 Pour Y allant de 0 à YM
 Ecran(X, Y) <-- nombre entier pris au hasard entre 0 et 15
```

```
FinPour Y
FinPour X
Fin

Module RECOPIE
(version 1)
Début
Pour X allant de 0 à XM
Pour Y allant de 0 à YM
DessinePixel(Ecran(X, Y))
FinPour Y
FinPour X
Fin

Module RECOPIE
(version 2)
Début
Pour Y allant de 0 à YM
Pour X allant de 0 à XM
DessinePixel(Ecran(X, Y))
FinPour X
FinPour Y
Fin
```

J'ai prix la liberté de remplacer les TantQue...FinTantQue par Pour...FinPour mais ça ne devrait pas vous poser de problème.

Le module HASARD affecte à chaque élément du tableau sans exception (grâce à l'utilisation de 2 boucles imbriquées) une valeur prise au hasard entre 0 et 15. Le module RECOPIE a pour fonction de recopier le contenu du tableau Ecran vers l'écran graphique.

La gestion du graphisme dépend énormément de la plateforme utilisée. Nous adopterons les conventions suivantes :

- les pixels sont numérotés de 0 à 319 horizontalement et de 0 à 199 verticalement
- le point (0,0) est le coin supérieur gauche de l'écran et le point (319,199) est le coin inférieur droit.
- les constantes XM et YM contiennent respectivement les résolutions horizontales et verticales de l'écran graphique. Ces constantes ont pour autre intérêt de faciliter la maintenance de votre programme. En effet, en supposant que vous vouliez utiliser une autre résolution graphique, vous n'avez alors qu'à modifier XM et/ou YM ainsi que la procédure d'initialisation du mode graphique et le tour est joué.
- les pixels sont repérés par deux numéros qui sont ses coordonnées horizontale et verticale.

Si la boucle Y correspondant aux coordonnées verticales est à l'intérieur de la boucle X (qui correspond aux coordonnées horizontales), comme dans le module RECOPIE Version 1, on verra le tableau se recopier colonne par colonne. Dans le module RECOPIE Version 2, le tableau est recopié ligne par ligne. Pour mieux saisir la subtilité de la chose faites-vous de petits programmes du genre :

```
Pour X allant de 10 à 30
 Pour Y allant de 1 à 10
 Afficher X, Y
 FPour Y
```

```
FPour X
```

D'accord cet exemple ne brillait pas par son originalité. Il a le mérite de ne pas être trop difficile à comprendre, en tout cas je l'espère.

Problème 2 : Mode Multi-image

Prendre une image de 320 sur 200 pixels et recomposer ses pixels pairs et impairs en 4 photos de 160 sur 100 pixels selont la répartition suivante :

```
Pairs ! Pairs
---------------------------------
Pairs  !  1  !  2  ! Impairs
----------!-----!-----!-----------
Pairs  !  3  !  4  ! Impairs
---------------------------------
Impairs ! Impairs
```

L'écran graphique est décomposé en 4 zones de taille égale composées comme suit : La zone 1 est composée des pixels horizontaux et verticaux de numéro pair (0, 2, 4, etc...). La zone 2 est composée des pixels horizontaux pairs et verticaux impairs. Pour la zone 3 la répartition est inverse de la zone 2. Enfin, la zone 4 est composée des pixels horizontaux et verticaux impairs.

En relançant la décomposition plusieurs fois de suite sur la même image on obtient des effets amusants. A vous d'essayer.

Ce qui me plaît dans ce problème, c'est le fait qu'il n'y a pas perte de l'information image. En effet, tous les pixels de l'image d'origine sont présents sur l'image de destination (mais pas à la même place). On pourrait pour s'en convaincre, s'amuser à écrire un algorithme qui reconstituerait l'image d'origine à partir des zones 1 et 4 ou à partir des zones 2 et 3.

```
Programme principal

 Constantes :
 XM = 320
 YM = 200

 Variables :
 IMAG : Tableau(0 à XM-1, 0 à YM-1)

 Début
 Faire initialise_Outils
 Faire chargement_Image
 Faire LECTURE(0, 0, XM-1, YM-1)
 Faire MULTI
 Faire fermeture_Outils
 Fin du module MULTIMAGE
```

Le programme principal est un peu fantaisiste quant à sa structure... peu importe, il est là à titre indicatif, à vous de l'adapter selon votre plateforme de développement.

```
Procédure LECTURE(A1, B1, A2, B2 : paramètres de type nombre entier)
```

```
 Variables locales :
 A, B : variables locales de type nombre entier

 Début
 Pour A allant de A1 à A2
 Pour B allant de B1 à B2
 IMAG(A, B) <-- PrendPixel(A, B)
 FinPour B
 FinPour A
 Fin de la procédure LECTURE
```

La fonction PrendPixel ci-dessus est une fonction qui retourne la couleur du pixel en position (A, B) ce qui permet de la stocker dans le tableau IMAG. La fonction DessinePixel ci-dessous est une fonction qui dessine un pixel en fonction des paramètres : position horizontale, position verticale et couleur.

```
Module MULTI

 Variables locales :
 XC, YC, AX, BX, AY, BY, PPX, IPX, PPY, IPY : variables locales
 de type nombre entier
 Début
 XC <-- XM DIV 2
 YC <-- YM DIV 2
 BX <-- 1
 PPX <-- 0
 IPX <-- XC
 Pour AX allant de 0 à XM-1 avec un Pas de 2
 BY <-- 1
 PPY <-- 0
 IPY <-- YC
 Pour AY allant de 0 à YM-1 avec un Pas de 2
 Faire DessinePixel(IMAG(AX, AY), PPX, PPY)
 Faire DessinePixel(IMAG(AX, BY), IPX, PPY)
 Faire DessinePixel(IMAG(BX, AY), PPX, IPY)
 Faire DessinePixel(IMAG(BX, BY), IPX, IPY)
 BY <-- AY + 1
 PPY <-- PPY + 1
 IPY <-- IPY + 1
 FinPour AY
 BX <-- AX + 1
 PPX <-- PPX + 1
 IPX <-- IPX + 1
 FinPour AX
 Fin du module MULTI
```

Tout se joue dans l'incrémentation judicieuse des variables AX, AY, BX, BY (correspondant à la position du pixel d'origine) et PPX, PPY, IPX et IPY (correspondant à la position du pixel d'arrivée) et dans leur emploi comme paramètres passés à la procédure DessinePixel. Le premier appel à la procédure DessinePixel a pour but de tracer le pixel dans la zone 1, le second appel trace le pixel correspondant à la zone 2 et ainsi de suite. Vous remarquerez qu'il n'y a pas ici de calculs redondants. Vous pouvez réécrire les appels à la procédure DessinePixel en supprimant les variables BX et BY afin de bien comprendre le fonctionnement de la boucle si vous le jugez nécessaire.

Problème 3 : Rotation d'image

Comme son nom l'indique, cet algo effectue la rotation d'une image. L'utilisateur que vous êtes doit entrer l'angle de rotation en degré (de 1 à 359°, en effet à quoi bon effectuer une rotation de 0 ou de 360°) puis le taux de réduction de l'image en % (de 0 à 99). La rotation s'effectue autour du centre de l'image.

Le principe de la rotation est le suivant :

X = Cosinus(angle) * x + Sinus(angle) * y

Y = -Sinus(angle) * x + Cosinus(angle) * y

La procédure LECTURE sert à transférer l'image vers le tableau afin de conserver la couleur d'origine des pixels pour pouvoir les affecter à leur nouvelle position (c'est la même que celle vue plus haut, de même que la procédure DessinePixel). Je ne donne pas ici l'algo du programme principal.

N.B. : DEG et TX sont des variables globales tandis que XM et YM sont les constantes globales vues plus haut. On utilise ici aussi le tableau IMAG.

```
Module ROTATION

 Début
 Afficher 'Entrez le degré de rotation (1 à 359) :'
 Lire DEG
 Afficher 'Entrez le taux de réduction en % (0 à 99) :'
 Lire TX
 Faire LECTURE(0, 0, XM-1, YM-1)
 Faire effacement_écran
 Faire TOURNER
 Fin du module ROTATION


 Module TOURNER
Variables locales :
 ANG, A, B, C, D, XINTER, YINTER : de type nombre réel
 XC, YC, PX, PY : de type nombre entier

 Début
 XC <-- XM DIV 2
 YC <-- YM DIV 2
 ANG <-- DEG * PI / 180
 A <-- (1 - (TX / 100)) * COS(ANG)
 D <-- A
 B <-- (1 - (TX / 100)) * SIN(ANG)
 C <-- -B
 X <-- 0
 TantQue X <= XM-1
 Y <-- 0
 TantQue Y <= YM-1
```

```
XT <-- ARRONDI(A * (X - XC) + B * (Y - YC) + XC)
YT <-- ARRONDI(C * (X - XC) + D * (Y - YC) + YC)
Faire DessinePixel(IMAG(X, Y), XT, YT)
Y <-- Y + 1
FinTantQue
X <-- X + 1
FinTantQue
Fin du module TOURNER
```

Ce type de transformation d'image est disponible sur de nombreux programmes de dessin.

Problème 4 : le Lissage d'image

J'aurais peut être dû appeler ça "flou artistique" car en fait de lissage, ça donne dans la pratique des effets de flou assez amusants. Cela n'a pas grand chose à voir avec les puissantes fonctions d'anti-aliasing que l'on trouve sur certains logiciels de dessin. Le principe est en fait très simple : l'utilisateur entre un niveau de priorité sur le pixel courant qui est en fait un vulgaire coefficient (que nous appellerons C). Le programme parcourt tous les pixels de l'écran et, pour chaque pixel, fait la somme des couleurs des 8 pixels qui l'encerclent en affectant un coefficient 1 à chacune de ces couleurs, puis ajoute la couleur du pixel considéré en lui affectant le coefficient C.

```
(1)(1)(1)
 (1)(C)(1)
 (1)(1)(1)
```

Puis le programme fait la moyenne des 8+C couleurs afin d'obtenir la nouvelle couleur qui sera affectée au pixel considéré.

Le terme de lissage est exagéré mais on obtient cependant des effets assez jolis (en faisant varier le niveau de priorité) comme si on regardait l'image au travers d'un verre dépoli (ça dépend aussi de la palette de couleur utilisée par l'image).

```
Module LISSAGE

 Début
 Afficher 'Entrez le niveau de priorité du pixel courant (1 à 20) :'
 Lire PRIO
 PRIO <-- PRIO - 1
 Faire LECTURE(0, 0, XM-1, YM-1)
 Faire LISSER(1, 1, XM-2, YM-2)
 Fin du module LISSAGE


 Procédure LISSER(AX, AY, BX, BY : paramètres de type nombre entier)

 Variables locales de type nombre entier : X, Y, COL

 Début
 Pour X allant de AX à BX
 Pour Y allant de AY à BY
 COL <-- IMAG(X, Y) * PRIO + LISSE(X-1, Y-1, X+1, Y+1)
 Faire DessinePixel(ARRONDI(COL / (9 + PRIO)), X, Y)
```

```
FinPour Y
FinPour X
Fin de la procédure LISSER


Fonction LISSE(PX1, PY1, PX2, PY2 : nombres entiers) : nombre entier

Variables locales de type nombre entier: PX, PY

Début
Pour PX allant de PX1 à PX2
Pour PY allant de PY1 à PY2
LISSE <-- LISSE + IMAG(PX, PY)
FinPour PY
FinPour PX
Fin de la fonction LISSE
```

Voilà une bien belle occasion de réviser les fonctions, pas vrai ? Je vous conseille de vous attarder sur cet algo, de l'éplucher et de bien le comprendre. Ce n'est pas tant ce qu'il fait que la façon de le faire qui est intéressante. Observez les appels aux procédures LISSER et LISSE.

Problème 5 : le Zoom

Bon d'accord, on en trouve sur tous les logiciels de dessin. Oui mais un zoom qui donnerait à l'utilisateur la possibilité de choisir des grossissmements horizontaux et verticaux différents, on en trouve pas à tous les coins de rue.

On commence par demander à l'utilisateur les coordonnées de la zone à zoomer. Ces coordonnées repérées par les variables X1, Y1, X2 et Y2 sont agencées comme suit : X1 et Y1 localisent le coin supérieur gauche de la portion de l'image à zoomer et X2 et Y2 localisent le coin opposé. La saisie de la zone à zoomer pourrait se faire à la souris (ce serait certainement plus pratique). Puis l'utilisateur entre les grossissements horizontal et vertical de la zone saisie. A noter qu'un grossissement de 1 équivaut à pas de grossissement du tout, ce qui permet de ne grossir l'image que dans un sens si on veut. Je rappelle que les grossissements horizontal et vertical peuvent être différents. Le programme recalcule X2 et Y2 pour éviter d'avoir à zoomer une image dont une trop grande partie serait invisible à l'écran. Prenons un exemple. Supposons que X1 = 10, Y1 = 30, X2 = 200 et Y2 = 100. Supposons que le grossissement soit de 2 dans les deux sens. Après le zoom, l'image aura la taille suivante :

- horizontalement : (200 - 10) * 2 = 380
- verticalement : (100 - 30) * 2 = 140

Verticalement, ça passe car la taille maximale de l'écran est de 200 pixels, mais horizontalement... y'a comme un défaut. On recalcule donc X2 comme suit : 10 + 320/2 - 1 = 169.

Une fois ces calculs terminés, on recopie avec le module COPIZOOM la zone d'image à zoomer dans le tableau à partir de la position 0,0. Le zoom se fait par un jeu d'incrémentations de variables (cf. procédure Zoomer). Chaque pixel est "remplacé" par un rectangle mis à l'échelle du grossissement et rempli avec la couleur du pixel qui lui correspond. Pour le tracé du rectangle, je fais appel aux routines de QuickDraw (cf. procédure RECTANGLE) qui sont très rapides. Il serait bête de s'en priver.

```
Module ZOOM
```

```
Variables Globales de type nombre entier :
X1, Y1, X2, Y2, PAS1, PAS2, XI, YI

Début
Afficher 'Coordonnées de la zone à zoomer'
Lire X1, Y1, X2, Y2
Afficher 'Entrez le grossissement horizontal en pixel (de 1 à 9) :'
Lire PAS1
Afficher 'Entrez le grossissement vertical en pixel (de 1 à 9) :'
Lire PAS2
Si (X2 - X1) > (XM DIV PAS1)
Alors X2 <-- X1 + ARRONDI(XM / PAS1) - 1
FinSi
Si (Y2 - Y1) > (YM DIV PAS2)
Alors Y2 <-- Y1 + ARRONDI(YM / PAS2) - 1
FinSi
Faire COPIZOOM
Faire effacement_écran
Faire ZOOMER
Fin du module ZOOM


Module COPIZOOM

Variables locales de type nombre entier : X, Y

Début
XI <-- 0
Pour X allant de X1 à X2
YI <-- 0
Pour Y allant de Y1 à Y2
IMAG(XI, YI) <-- PrendPixel(X, Y)
YI <-- YI + 1
FinPour Y
XI <-- XI + 1
FinPour X
Fin du module COPIZOOM


Module ZOOMER

Variables locales de type nombre entier : PX, PY, X, Y

Début
PX <-- 0
Pour X allant de 0 à (XI - 1)
PY <-- 0
Pour Y allant de 0 à (YI - 1)
Faire RECTANGLE(IMAG(X, Y), PX, PY, PX + PAS1, PY + PAS2)
PY <-- PY + PAS2
FinPour Y
PX <-- PX + PAS1
```

```
FinPour X
Fin du module ZOOMER
```

Quelques calculs redondants sont là-aussi à supprimer.

Problème 6 : la Réduction

Cet algo permet de réduire une image en choisissant le rapport de réduction en pourcentages (de 0 à 99, car demander une réduction de 100 % me parait ridicule), ce rapport n'étant pas forcément le même sur les axes horizontal et vertical. On peut choisir par exemple une réduction de 50 % sur l'axe horizontal et une réduction de 20 % sur l'axe vertical. Si vous demandez une réduction de 50 %, l'ordinateur sautera un pixel sur deux, c'est à dire qu'il n'affichera que les pixels 0,2,4,6,etc..., sans espace entre ces pixels (ce qui donnera l'effet de réduction souhaité).

```
Module REDUCTION

 Variables Globales de type nombre entier :
 X1, Y1, X2, Y2, PAS1, PAS2, XMAX, YMAX

 Variables locales de type nombre entier : CENT1, CENT2

 Début
 Afficher 'Coordonnées de la zone à zoomer'
 Lire X1, Y1, X2, Y2
 Afficher 'Entrez le taux de réduction horizontal en % (0 à 99)'
 Lire CENT1
 Afficher 'Entrez le taux de réduction vertical en % (0 à 99)'
 Lire CENT2
 PAS1 <-- 1 / ((100 - CENT1) / 100)
 PAS2 <-- 1 / ((100 - CENT2) / 100)
 Faire REDUIRE
 Faire effacement_écran
 Faire AFFICHER
 Fin du module REDUCTION


 Module REDUIRE

 Variables locales :
 X, Y : de type nombre Réel (oui, vous avez bien lu)

 Début
 XMAX <-- 0
 X <-- X1
 Répéter
 YMAX <-- 0
 Y <-- Y1
 Répéter
 IMAG(XMAX, YMAX) <-- PrendPixel(ARRONDI(X), ARRONDI(Y))
 Y <-- Y + PAS2
 YMAX <-- YMAX + 1
 Jusqu'à Y > Y2
```

```
X <-- X + PAS1
XMAX <-- XMAX + 1
Jusqu'à X > X2
Fin du module REDUIRE
```

Quelques calculs redondants sont là-aussi à éliminer.

```
Module AFFICHER

Variables locales de type nombre entier : TX, TY, X, Y

Début
TX <-- 0
Pour X allant de 0 à XMAX
TY <-- 0
Pour Y allant de 0 à YMAX
Faire DessinePixel(IMAG(X, Y), TX, TY)
TY <--TY + 1
FinPour Y
TX <-- TX + 1
FinPour X
Fin du module AFFICHER
```

Vous remarquerez qu'on n'utilise pas ici la procédure LECTURE car on fait les calculs et les affectations vers le tableau IMAG dans la même procédure (cf. REDUIRE), la recopie de l'image réduite, du tableau vers l'écran, se fait ensuite dans une procédure indépendante (cf. AFFICHER).

On pourrait écrire un module qui ferait la réduction par "pondération de couleurs" ce qui signifie que pour une réduction de 50 %, l'ordinateur prendrait la couleur des pixels 0 et 1, l'additionnerait et en ferait une moyenne qui serait la couleur du pixel 0 après réduction (et ainsi de suite pour les autres pixels) : une sorte de mixage entre réduction et lissage, quoi. Il est évident que ce mode de réduction serait un peu plus lent que l'autre (qu'on pourrait alors appeller réduction par "omission de pixels", par exemple). Quel que soit le mode de réduction choisi, c'est le module AFFICHER qui se chargerait du transfert du tableau vers la pixel map. Si le coeur vous en dit...

En conclusion :

On n'a pas vu grand chose de nouveau cette fois-ci si ce n'est les tableaux à 2 dimensions. Nous avons revu des notions telles que les fonctions et procédures avec ou sans passages de paramètres, les variables locales et globales, et les boucles. Les algorithmes sont suffisamment généraux pour être adaptés sans trop de mal à tout environnement de programmation. Si je n'ai pas de moi-même supprimé tous les calculs redondants, c'est que je craignais de rendre les algos par trop illisibles pour le lecteur.

Si vous êtes arrivé jusqu'à ces lignes sans trop de mal, c'est que vous n'êtes déjà plus un débutant, mais plutôt un amateur confirmé. C'est surtout que vous aurez fourni un sacré travail personnel et je vous en félicite. Si vous avez du mal avec ce cours, vous n'en avez pas moins de mérite, mais vous avez encore besoin de travailler et de rôder certaines notions.

Problème 7 : Le Placage sur une sphère

Les infographistes parleraient de "mapping" ce qui est quand même plus joli que placage. Il s'agit ni plus ni moins que de faire épouser aux formes d'un objet (en l'occurrence une sphère) une image sélectionnée au préalable. On indique le rayon en pixel de la sphère (de 10 à 100) et le degré d'inclinaison de l'image qui sera appliquée sur la sphère (de 0 à 359°). Les temps de calcul sont longs. Ne me demandez surtout pas d'expliquer les calculs de projection, parce que les maths et moi... ça fait vraiment 2. J'ai obtenu les formules de projection d'un ami (merci Didier) qui lui-même les tenait d'un prof de maths. Je conseille aux utilisateurs de choisir un rayon entre 70 et 90 ce qui donne des résultats satisfaisants mais ne permet malheureusement pas d'éviter une certaine distorsion de l'image comme vous pourrez le constater. Il sera souvent nécessaire de retoucher l'image avec un programme de dessin de type GS.Paint. N.B. : le placage de fractales, telles que certaines portions de l'ensemble de Mandelbrot, donne de très beaux résultats.

```
Module PLACAGE

 Variables glogales de type nombre entier : RAY, DEG

 Début
 Afficher 'Entrez le rayon de la sphère (10 à 100) :'
 Lire RAY
 Afficher 'Entrez le degré d'inclinaison de la sphère (0 à 359) :'
 Lire DEG
 Faire LECTURE(0, 0, XM-1,YM-1)
 Faire effacement_écran
 Faire SPHERE
 Fin du module PLACAGE
```

```
Module SPHERE

 Variables locales :
 PX, PY, CLA, A, B, AB, E, PIA, PIB, PIC, PROF, INCLIN : nombres réels
 XC, YC, X, Y : nombres entiers

 Début
 XC <-- XM DIV 2
 YC <-- YM DIV 2
 PIA <-- PI / 180
 PIB <-- PI / 2
 PIC <-- PI / 32
 INCLIN <-- DEG * PI / 180
 Pour X allant de X1 à X2
 PX <-- (350 - X) * PIA + 3
 Pour Y allant de Y1 à Y2
 PY <-- (YC - Y) * PIA
 CLA <-- COS(PY)
 PROF <-- CLA * COS(PX)
 Si PROF > 0 Alors
 A <-- SIN(PY)
```

```
B <-- CLA * SIN(PX)
AB <-- SQR(A * A + B * B) * RAY
E <-- ATN(A / B) - INCLIN + PIB * (1 - B / ABS(B))
Faire DessinePixel(IMAG(X, Y), ARRONDI(XC-COS(E)*AB),
ARRONDI(YC-SIN(E)*AB))
FinSi
FinPour Y
FinPour X
```

```
Fin du module SPHERE
```

Avant de nous quitter, parlons rapidement des tableaux à 3 dimensions. Vous vous demandez peut être à quoi ils servent. Je vais donc prendre quelques exemples.

Imaginez le gestionnaire d'une chaine de magasins qui veut effectuer des traitements statistiques tels que des comparaisons sur la rentabilité de différents magasins, rayons et produits. On pourra utiliser un tableau comme celui-ci :

Ventes : Tableau(1 à M, 1 à R, 1 à P) de nombres réels

- M est le nombre maximum de magasins
- R est le nombre maximum de rayons par magasin
- P est le nombre maximum de produits différents par rayon

Plusieurs méthodes toutes très honorables existent pour stocker le montant des ventes dans le tableau Ventes. Ca dépend aussi de la façon dont sont récoltées les informations.. Chaque magasin communique-t-il ses résultats sous forme de fichier sur disque ou sous forme d'imprimés ? Cette question toute simple justifierait un cours à elle-seule.

Supposons maintenant que notre gestionnaire veuille obtenir un listing présentant le résultat des ventes dans tous les magasins d'un produit qui se trouve dans un rayon particulier. On pourra écrire quelque chose comme ça :

```
...
Lire Prod, Rayon
Faire EditionProduit(Prod, Rayon)
...
Procédure EditionProduit(Z, Y)
Début
Pour X allant de 1 à M
Imprimer Ventes(X, Y, Z)
FinPour
Fin
```

Notre gestionnaire peut aussi vouloir des résultats en pourcentages, il peut vouloir comparer pour tous les magasins le montant total des ventes d'un rayon particulier sur plusieurs années, etc... Autant de besoins particuliers notre gestionnaire aura, autant de problèmes particuliers notre programmeur aura. Une bonne analyse du problème sera nécessaire, analyse qui devra être suivie de l'écriture d'algorithmes performants. Si après coup, notre gestionnaire s'aperçoit qu'il a oublié certains traitements statistiques dont il aurait besoin (ça arrive, vous pouvez me croire) le programmeur devra pouvoir y répondre rapidement, d'où l'intérêt d'une bonne méthode de programmation, d'où l'intérêt de l'algorithmique.

Les tableaux à trois dimensions (ou plus) servent aussi à représenter des images en 3 dimensions telles que des polyèdres ou toutes sortes d'objets qu'on a besoin de visualiser sous plusieurs angles. Ils peuvent aussi servir à modéliser des cubes de données telles que des chiffres d'affaire (coordonnée X) par article (coordonnée Y) et par année (coordonnée Z). Tout un programme... Quelquefois, l'informatique de gestion et l'informatique graphique se rejoignent.

Chapitre 9 - Récréations

Problème 1 :

On veut écrire un programme qui tracera un arc-en-ciel en mode graphique (320 sur 200 pixels) sur l'écran de notre micro préféré. Il aura à peu près la forme suivante (en mode texte, c'est pas facile à représenter) :

```
------------------
 ! / / / / / / / /!
 !/ / / / / / / / !
 ! / / / / / / / /!
 !/ / / / / / / / !
 etc... (jusqu'en bas de l'écran)
```

On va donc saucissoner notre écran en 16 tranches obliques (un nombre pair est plus pratique pour le calcul des coordonnées), chacune de ces tranches devant être remplie avec une couleur différente (ça tombe bien on a justement 16 couleurs à notre disposition).

Je vous l'accorde, il serait plus vite fait de dessiner cet arc-en-ciel avec un programme de dessin. Et alors, seriez-vous devenu paresseux ? J'y pense, ici on n'aura pas besoin de tableaux, alors ne prenez pas un fusil à pompe pour aplatir un moustique.

```
Programme principal

 Début
 Faire initialise_tools
 Faire arc_en_ciel
 Faire extinction_des_feux
 Fin.

 Module initialise_tools
 Module extinction_des_feux
```

Ces deux modules sont spécifiques à chaque environnement de développement quant à la syntaxe. Le premier module aura pour but d'initialiser le mode graphique, le second de fermer les outils initialisés par le premier. Reportez-vous au cours sur la Toolbox dans GS.Infos pour savoir comment initialiser les tools et dans quel ordre. Vous pouvez très bien vous passer de la Toolbox pour utiliser le mode Fill, alors je ne m'appesantirai pas davantage, ce n'est pas l'objet de mon cours.

Avant d'attaquer le module arc_en_ciel, passons en revue les problèmes qui se posent :

- le remplissage de l'écran : on va se servir du mode "Fill" de notre Apple IIGS pour remplir chaque tranche de l'écran. Dans ce mode, chaque point de couleur 0 prend la couleur du point qui le précède. Ce qui veut dire qu'on perd l'usage de la couleur 0 qui est noire. Il nous reste donc 15 couleurs numérotées de 1 à 15. Il faudra feinter pour répartir les 15 couleurs harmonieusement dans les 16 tranches. Ainsi,

dans l'algo ci-dessous, vous remarquerez que la couleur 8 est "appelée" par deux fois. Pour activer le mode Fill, on doit mettre le bit 5 du SCB (Scan line Control Byte) de chaque ligne à 1. Avec QuickDraw (toolbox graphique de l'AppleIIGS), on peut activer le mode Fill de toutes les lignes en même temps avec SetAllScbs(argument). Dans le cas présent, l'argument est égal à 32 car 2 exposant 5 = 32. Pour le fonctionnement des SCBs, voir la doc Apple. Dans ce mode, la couleur du premier pixel de chaque ligne doit être différente de 0, sinon on risque d'avoir des effets indésirables, ce qui explique la présence de la Boucle 1. Supprimez-la à l'occasion, l'effet est plutôt joli, finalement. Si vous avez du mal à comprendre le fonctionnement du mode Fill, ou le fonctionnement de l'algo lui-même, supprimez les SetAllScbs() et relancez le programme.

- le dessin lui-même : l'écran étant découpé en 16 tranches, il va falloir calculer les coordonnées limites de chaque segment de droite. Une ligne contient 320 pixels. En colonne, on a 200 pixels. Puisque nos segments sont tracés

Post-scriptum :

Cliquer ici pour les premiers chapitres.