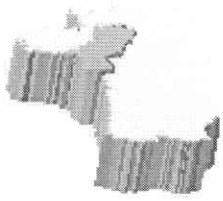
IREM

INSTITUT DE RECHERCHE SUR L'ENSEIGNEMENT DES MATHEMATIQUES
DES PAYS DE LA LOIRE



NANTA IREMICA N° 14

Mathématiques et langage informatique

Tome 1

(basic)

J. Betrema

1978

UNIVERSITE DE NANTES

I. R. E. M. DE NANTES

◆◆

MATHEMATIQUES ET LANGAGE INFORMATIQUE (BASIC)

◆◆◆

VOLUME I

par J. BETREMA

◆ NANTA IREMICA

Volume N° 14

JANVIER 1978 ◆

AVERTISSEMENT

Depuis plusieurs années arrivent sur le marché les mini-ordinateurs, ou même les ordinateurs portables, construits autour de microprocesseurs. Leurs prix baissent aussi vite que leur taille, tandis qu'au contraire leurs performances s'améliorent considérablement. En particulier, on trouve sur les ordinateurs portables des langages évolués ; BASIC est probablement le plus répandu.

Cette brochure est issue d'un travail important effectué à l'IREM de Nantes pour permettre aux professeurs de mathématiques de se familiariser avec l'informatique, et d'étudier les relations entre informatique et enseignement des mathématiques. Les IREM et l'INRP ont souligné depuis longtemps, et à juste titre, combien la présence d'un ordinateur dans une classe serait susceptible de transformer l'enseignement. Et d'abord parce que l'ordinateur, qui peut répondre à l'utilisateur, modifie profondément l'environnement des élèves (ou de stagiaires en formation permanente) : l'enseignant cesse d'être le seul interlocuteur. D'où, on peut l'espérer, de nouvelles possibilités d'autonomie et d'attitude active d'apprentissage pour les enseignés sauf, bien sûr, si un jour on transforme l'informatique en une nouvelle matière d'examen.

Cependant, pour que l'utilisateur puisse dialoguer avec l'ordinateur, il doit acquérir un langage informatique, car un dialogue du genre :

Utilisateur : "Pourriez-vous me donner les racines de l'équation

$$x^2 - 3x + 2 = 0 \text{ ?"}$$

Ordinateur : "Bien sûr ; je trouve $\{1, 2\}$ comme ensemble de solutions" n'est pas pour demain (d'ailleurs ce jour-là il faudra vraiment commencer à s'inquiéter).

Cette brochure vise à présenter aux lecteurs mathématiciens, à travers l'apprentissage d'un langage répandu, BASIC, quelques-uns des concepts fondamentaux qui, aujourd'hui, semblent nécessaires pour parler à un ordinateur, c'est-à-dire pour le programmer. La brochure insiste sur la structure des programmes, car l'expérience montre que la programmation peut être conçue comme une activité systématique, et non comme une sorte de bricolage futuriste réservé à des êtres étranges dont le cerveau aurait fini par ressembler à un ordinateur.

Nous n'avons besoin ni de spécialistes familiers de toutes les astuces d'emploi d'un langage de programmation donné, ni, corrélativement, de gens pour qui écrire un

programme reste une aventure si périlleuse qu'ils préfèrent la laisser à ceux "qui ont la bosse". Programmer doit être une activité suffisamment simple pour qu'en un temps raisonnable on puisse obtenir de l'ordinateur les résultats attendus. Pour cela, il est nécessaire d'acquérir un bon style dans l'écriture des programmes ; on a depuis quelques années, à la suite de travaux d'informaticiens comme DIJKSTRA ou WIRTH, une idée plus précise de ce que peut être un programme bien structuré. Il est regrettable que ces idées ne soient pas encore aussi répandues qu'elles devraient l'être dans les IREM ; peut être parce que souvent les animateurs IREM en informatique se sont seulement formés, nécessité oblige, au contact d'une machine disponible.

Il est utile de lire de "bons" programmes en même temps que d'en écrire. Cette brochure présente un choix assez important de programmes simples et intéressants pour un mathématicien (alors que les ouvrages d'introduction au BASIC qu'on trouve dans le commerce s'adressent souvent à des gestionnaires). Le lecteur aura évidemment intérêt, après avoir lu quelques exemples, à essayer d'écrire lui-même les suivants, et à comparer sa solution avec celle proposée dans le texte (qui n'est pas forcément la meilleure ...) BASIC, contrairement à une opinion répandue, ne convient que médiocrement à une initiation à la programmation :

- la structure du langage, à base d'instructions de branchement, ne favorise guère la compréhension des schémas de programmes fondamentaux. Cette difficulté a été en partie tournée dans cette brochure en introduisant, pour le développement des algorithmes, un langage fictif (cf. section 4.4), très dépouillé, dont les caractéristiques ont été empruntées à MANNA (cf. bibliographie).

- BASIC ne comporte qu'une forme dégénérée de sous-programmes, si bien que cette brochure ne comporte aucune étude de questions pourtant fondamentales en informatique comme les variables locales, les paramètres d'une procédure, etc ...

Ceci dit, on peut faire un travail intéressant avec BASIC ; mais dès que les constructeurs mettront à notre disposition, sur des machines de taille réduite, de meilleurs langages, BASIC devra probablement être abandonnée. Déjà outre LSE, disponible sur MITRA 15 ou T 1600, APL,disponible sur IBM 5100, et LOGO, disponible par l'intermédiaire de la société québecquoise Général Tortue, sont à étudier de près. Inutile donc de se perdre dans l'étude trop détaillée d'un langage donné, comme BASIC ; en matière de langage informatique, il faut garder l'esprit très ouvert car dans ce domaine, les choses évoluent très vite, et c'est heureux ! Et pourquoi un jour les IREM ne se doteraient-ils pas d'une structure capable de créer, d'implanter sur micropocesseurs, et de maintenir vivant, un langage adapté à nos besoins ?

Les chapitres 1 et 2 sont des chapitres d'introduction ; le chapitre 1 présente la particularité de contenir un exemple de preuve de programme. Le chapitre 3 traite des entrées-sorties en BASIC : il peut être parcouru en diagonale en première lecture. Le chapitre 4 : boucles et tests constitue le coeur de la brochure.

Les chapitres suivants :

5 - Tableaux
6 - Sous-programmes
7 - Chaînes de caractères

sont relativement indépendants les uns des autres et peuvent être lus pour l'essentiel dans un ordre quelconque après le chapitre 4. Le chapitre 7 contient trois algorithmes de permutations non évidents.

Cette brochure sera suivie d'un volume 2, qui présentera des programmes un peu plus compliqués étudiés à l'IREM de Nantes ces trois dernières années ; les sujets couverts seront : analyse numérique, arithmétique, nombres aléatoires, méthodes de tri, graphes. L'accent sera mis sur les analyses d'algorithmes (étude mathématique de la complexité des calculs mis en oeuvre pour résoudre un problème donné), les méthodes de développement de programmes, et sur les structures de données.

J'espère que le lecteur y découvrira parfois des mathématiques d'un type nouveau. L'informatique contribue à l'étude de nouveaux problèmes mathématiques ; il ne faudrait pas que le volume 1 de cette brochure, ou d'autres publications des IREM, laissent croire au lecteur que l'informatique ne sert qu'a illustrer des notions traditionnelles, ou à produire quelques divertissements.

Les deux volumes de cette brochure pourront aider, je l'espère, les participants aux stages IREM orientés vers l'informatique ; mais je souhaite qu'ils puissent aussi être utiles de façon plus générale aux personnes à formation scientifique qui souhaitent savoir un peu précisément, et avec un minimum de cohérence, ce qu'est une procédure de calcul en informatique, quels sont les types de problèmes mathématiques relativement simples à traiter sur ordinateur, et quelles sont les méthodes de base pour passer de la formulation courante d'un problème à sa formulation informatique. Bien entendu, l'informatique permet de s'attaquer à des catégories de problèmes beaucoup plus vastes que ceux présentés ici, par exemple en Intelligence Artificielle, et il serait peut-être intéressant que les IREM s'intéressent à ces questions ; sinon nous aurons une vue

étroite des relations informatique -enseignement des mathématiques. Un groupe inter-IREM a d'ailleurs commencé à travaillé autour du projet LOGO développé au Massachusetts Institute of Technology.

Mes remerciements à tous mes collègues de l'IREM de Nantes et aux stagiaires avec qui j'ai travaillé ces trois dernières années ; aux animateurs du groupe inter-IREM intitulé "récursivité" ; à Madame ROBIN, secrétaire à l'IREM de Nantes, pour la qualité de son travail ; et, pour paraphraser KNUTH (cf. bibliographie), à l'IBM 5100 de l'IREM de Nantes, qui ne m'a pas ménagé son appui, même dans certains moments difficiles.

J. BETREMA

Novembre 1977

TABLE DES MATIERES

.../...

1 EUCLIDE

A quoi ressemble un langage informatique, en particulier le langage BASIC ? Comment parle-t-on à un ordinateur ? Qu'est-ce qu'un programme ? Nous souhaitons, dans ce premier chapitre, donner immédiatement un début de réponse aux lecteurs débutants en informatique. Ce sera en même temps l'occasion (déjà) d'aborder une contribution importante de l'informatique à l'activité mathématique : les preuves de programmes.

Nous prendrons comme exemple un programme, écrit en BASIC, qui calcule le PGCD de deux entiers naturels a et b.

§ 1-1 COMMENT UN ORDINATEUR PEUT APPRENDRE À CALCULER UN PGCD

La méthode employée est ancienne, puisqu'on l'appelle algorithme d'Euclide (algorithme : procédure automatique de calcul). Elle consiste en une suite de divisions (euclidiennes !).

$$
\text{(I)} \qquad \left|\begin{array}{l}
a = bq_0 + r_1 \\
b = r_1\, q_1 + r_2 \\
r_1 = r_2\, q_2 + r_3 \\
\quad\vdots \\
r_{n-2} = r_{n-1}\, q_{n-1} + \boxed{r_n} \\
r_{n-1} = r_n\, q_n
\end{array}\right.
$$

On arrête (et pour cause) cette suite de divisions dès qu'on trouve un reste nul ; le dernier reste non nul (désigné ci-dessus par r_n) est alors le PGCD de a et b (pour une justification de la méthode, voir plus loin).

Exemple : calcul du PGCD de 126 et 33

$126 = 33 \times 3 + 27$

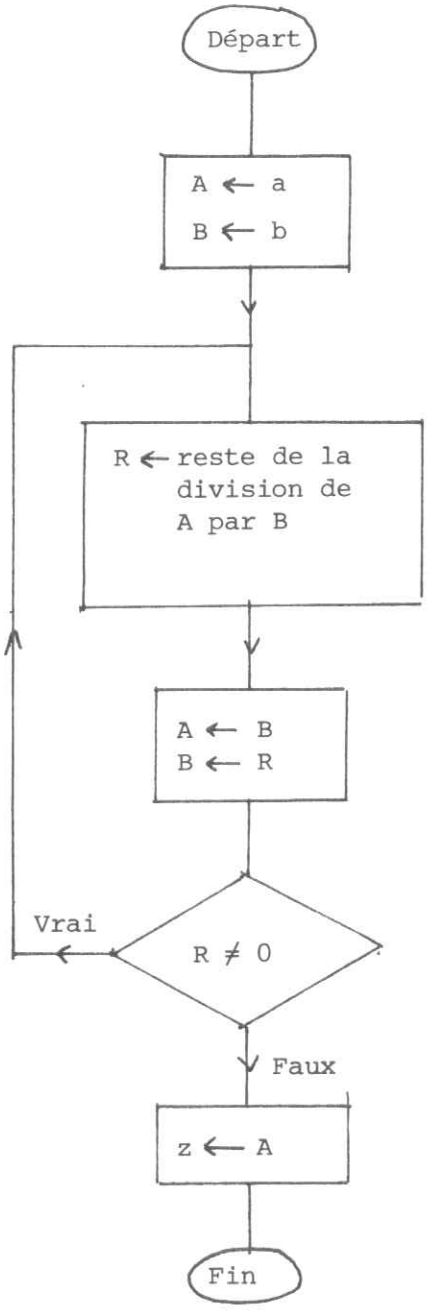
$33 = 27 \times 1 + 6$

$27 = 6 \times 4 + 3$

$6 = 3 \times 2$

Le dernier reste non nul, 3, est le PGCD cherché.

Cet algorithme peut être décrit plus précisément par l'organigramme suivant, dont nous allons ensuite préciser le sens :



Organigramme 1.1

Cet organigramme utilise trois variables de calcul : A, B, R. Le résultat du calcul est désigné par z† (ici z désigne donc le PGCD de a et b). Les flèches désignent des affectations ; par exemple si a = 126, l'instruction

$$A \longleftarrow a$$

signifie : affecter la valeur 126 à la variable A.

L'utilisation de variables présente le même avantage que partout ailleurs en mathématiques : on désigne ainsi d'un même nom un objet susceptible de prendre une infinité de valeurs.

L'organigramme 1.1 décrit la procédure euclidienne de calcul d'un PGCD. Cela signifie que si l'on part de la case Départ de l'organigramme, et si l'on suit les flèches en exécutant à chaque fois les instructions prescrites, la valeur finale attribuée à z est le PGCD des entiers a et b.

Exemple :

Exécutons l'organigramme 1.1 avec a = 126 et b = 33

- A reçoit la valeur 126

  B reçoit la valeur 33

- R reçoit la valeur 27

- A reçoit la valeur de B, c'est-à-dire 33

- B reçoit la valeur de R, c'est-à-dire 27

- On teste si la valeur actuelle de R est différente de zéro; comme la réponse est VRAI, on continue l'exécution de l'algorithme en suivant la branche marquée VRAI.

- R reçoit la valeur 6

- A reçoit la valeur 27

---

† Cette convention est valable pour l'ensemble de l'ouvrage.

- B reçoit la valeur 6

- Test : R $\neq$ O ? VRAI
- R reçoit la valeur 3
- A reçoit la valeur 6
- B reçoit la valeur 3

- test : R $\neq$ O ? VRAI
- R reçoit la valeur O
- A reçoit la valeur 3
- B reçoit la valeur O

- test : R $\neq$ O ? FAUX

On suit donc la branche marquée FAUX : z (qui désigne le résultat) reçoit la valeur 3, et l'exécution prend fin.

Nous espérons que le lecteur est convaincu que l'organigramme 1.1 est bien une description de l'algorithme d'Euclide.

Cette description par organigramme est en fait plus simple ; en effet ;

- l'organigramme 1.1 n'utilise que trois variables de calcul ; les indices de l'écriture

$$
(I) \quad \left|\begin{array}{l}
a = bq_0 + r_1 \\
b = r_1\, q_1 + r_2 \\
r_1 = r_2\, q_2 + r_3 \\
\vdots \\
r_{n-2} = r_{n-1}\, q_{n-1} + r_n \\
r_{n-1} = r_n\, q_n
\end{array}\right.
$$

s'avèrent donc être une complication inutile.

- l'organigramme 1.1 n'utilise que deux types d'instructions : instructions d'affectation et instructions de test (outre les instructions (Départ) et (Fin)).

Utilisons l'organigramme 1.1 pour rappeler la justification mathématique de l'algorithme d'Euclide, c'est-à-dire pour prouver :

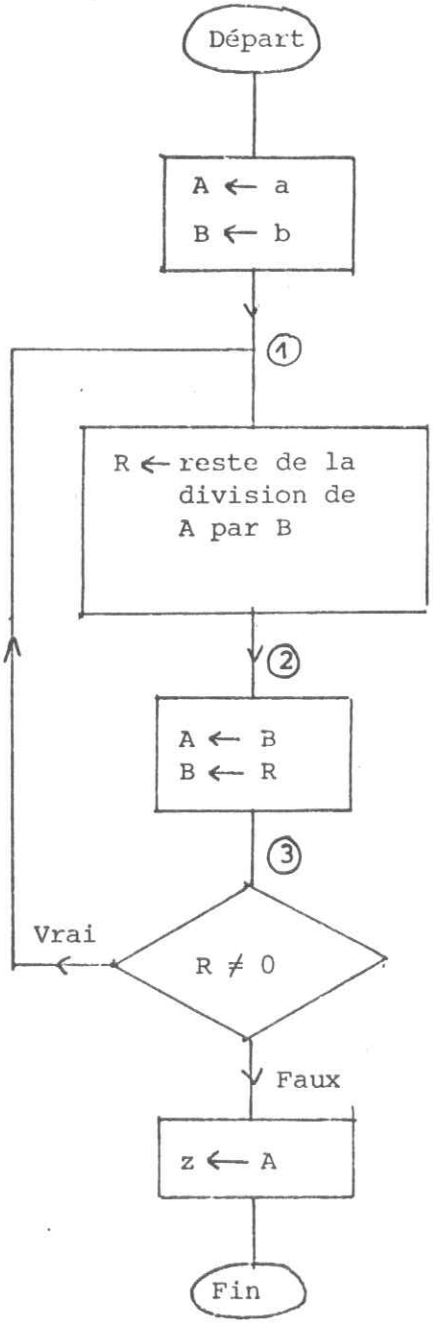
(i) que lorsqu'on atteint l'instruction (Fin) , z a pour valeur le PGCD de a et b.

(ii) que l'on atteint effectivement, c'est-à-dire au bout d'un temps fini, l'instruction (Fin) (pour tous entiers naturels a et b).

La décomposition en deux parties de cette preuve correspond aux deux types de programmes incorrects:

- programmes fournissant un résultat 'faux'.
- programmes ne fournissant pas de résultat du tout! Dans ce cas on dit que le programme boucle: son exécution consiste en la répétition infinie d'une certaine suite d'instructions.

La démonstration utilise l'organigramme 1.1, reproduit page suivante avec des numéros comme (1), (2), etc... qui servent à en identifier certains points.


Départ
A ← a
B ← b
①
R ← reste de la division de A par B
②
A ← B
B ← R
③
Vrai
R ≠ 0
Faux
z ← A
Fin

Démonstration de (i)

Lorsqu'en cours d'exécution de l'organigramme on atteint pour la première fois le point (1), on a : A = a et B = b , et donc:

(1) PGCD (A, B) = PGCD (a, b)

Montrons que cette formule reste vraie chaque fois qu'on passe par le point (1). L'argument évident donné ci-dessus servira de première étape pour un raisonnement par induction, dont la deuxième étape est :

> Si la formule (1) est vraie en (1), alors elle reste vraie après exécution de la boucle (1) - (2) - (3) - (1).

En d'autres termes, prouvons que la formule (1) est un invariant de la boucle :

Au point (2) on a: A = BQ + R où Q est un entier, donc tout diviseur commun de A et B divise R, et réciproquement tout diviseur commun de B et R divise A; donc:

PGCD (A,B) = PGCD (B,R)

Au cas où R = 0, cette formule reste vraie avec la convention:

PGCD (B,0) = B.

On passe ensuite de (2) à (3) en exécutant les instructions:

A ← B

B ← R

Puisque PGCD (A,B) = PGCD (B,R), ces instructions laissent donc inchangé PGCD (A,B) - avec la convention PGCD (A,0) = A -. Ceci prouve que si la formule (1) est vraie au point (1), elle reste vraie au point (3) (avec la convention PGCD (A,0) = A), et achève la preuve par induction annoncée.

Enfin, si l'on arrive à l'instruction (Fin), c'est en provenant de (3), avec B = R = 0. Donc la formule (1) devient:

A = PGCD (a,b)

et l'instruction: z ← A attribue à z la valeur attendue.

Démonstration de (ii).

Ce point est crucial: pourquoi l'algorithme d'Euclide ne mène-t-il pas à une suite sans fin de divisions ? En d'autres termes, pourquoi est-il astucieux (et non pas absurde) de ramener le calcul du PGCD de a et b à celui de b et r (où r désigne le reste de la division de a par b) ?

La raison est bien connue : le reste de la division euclidienne de A par B est strictement inférieur à B, donc lors de l'exécution de l'organigramme 1.1 les valeur de R sont des entiers naturels strictement décroissants : on atteint donc R = O au bout d'un temps fini. †

Nous avons donné cette preuve car elle est caractéristique d'une activité mathématique nouvelle, liée à l'essor de l'informatique : prouver qu'un algorithme est correct, c'est-à-dire fournit le résultat attendu (moyennant certaines conditions sur les données ; ici a et b doivent être des entiers naturels). Les arguments employés dans cette preuve ne sont pas, quand au fond, nouveaux ; mais leur présentation est nouvelle, et fait apparaître leur fonction exacte dans la preuve. Ce type de preuve est appelé 'preuve par assertions'.

L'organigramme 1.1 peut facilement être traduit en BASIC :

Programme PGCD

```
0100 INPUT A,B
0110 R=A-B*INT(A/B)
0120 A=B
0130 B=R
0140 IF R≠0 GOTO 0110
0150 PRINT A
0160 END
```

---

† Calculer lequel, c'est-à-dire le nombre de divisions, est une autre histoire : cf. chapitre 9

Comme on le voit, un programme BASIC est constitué d'une suite d'instructions numérotées †, régies par une syntaxe simple mais rigoureuse. Un programme BASIC peut être exécuté par quiconque comprend le BASIC ; si vous avez sous la main un ordinateur qui comprend le BASIC, vous pouvez essayer. Il faudra bien entendu commencer par entrer le programme, en le frappant sur un clavier relié à l'ordinateur. Puis on frappe : RUN et l'ordinateur exécute les instructions :

- 100 INPUT A,B

L'ordinateur affiche sur un écran (ou imprime sur un papier) un point d'interrogation, signalant ainsi qu'il attend que l'utilisateur entre des données. Répondez par exemple (toujours au moyen du clavier) :

126, 33

puis appuyez sur la touche "retour de chariot" pour signaler que votre réponse est complète††. L'ordinateur affecte la valeur 126 à un registre de mémoire, auquel il affecte aussi le nom A ; toute référence ultérieure à A sera interprétée comme désignant la valeur contenue dans ce registre ; en abrégé, "A reçoit la valeur 126" ; de même B reçoit la valeur 33.

L'instruction 100 est donc une forme d'instruction d'affectation, mais où les valeurs sont fournies par l'utilisateur (au lieu de résulter de calculs effectués par l'ordinateur) ; c'est pourquoi on la classe parmi les instructions d'entrée.

- 110 R = A-B * INT (A/B)

En BASIC, le signe d'affectation ⟵ est remplacée par =, ce qui est une assez mauvaise chose, car génératrice de confusions avec d'autres emplois du signe =.

---

† . On verra plus loin pourquoi on commence souvent par le numéro 100, et progresse de 10 en 10.

†† . Sinon l'ordinateur attend la suite, car 33 peut être interprété comme le début d'un nombre de trois chiffres ou plus.

A droite du signe = figure une expression qui est la traduction BASIC de "reste de la division euclidienne de A par B". Le signe * désigne la multiplication, le signe / la division, et INT la fonction partie entière.

L'instruction 110 peut donc se lire :

R ⟵ A - B × (quotient "exact" de A par B)

L'ordinateur, lorsqu'il exécute l'instruction 110, attribue donc la valeur 27 au registre désigné par R.

- 120 A=B

  130 B=R

Instructions d'affectation : A reçoit la valeur 33

B reçoit la valeur 27

- 140 IF R≠0 GOTO 110

L'ordinateur teste la valeur contenue dans le registre désigné par R ; comme le résultat du test est positif, il poursuit l'exécution du programme en retournant à l'instruction 110.

110 R=A-B * INT (A/B)

etc ...

L'exécution continue, exactement comme elle a été décrite précédemment (exécution de l'organigramme 1.1) jusqu'à ce que A reçoive la valeur 3 et B la valeur 0.

- 140 IF R≠0 GOTO 110

Cette fois le résultat du test est négatif ; l'ordinateur ignore donc l'instructio GOTO 110 et passe à l'instruction qui suit l'instruction de test.

- 150 PRINT A

est une instruction de sortie : l'ordinateur imprime un résultat. Sans instruction de sortie, l'ordinateur garde pour lui le résultat de ses calculs, ce qui n'est guère

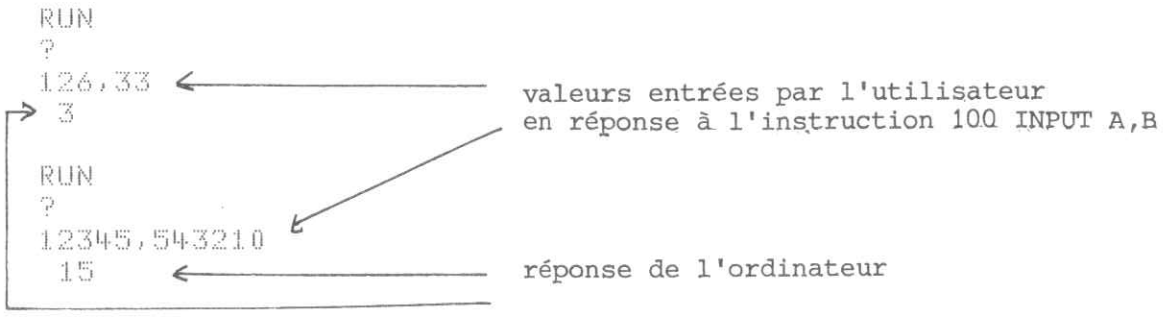
plaisant pour l'utilisateur. En effet, l'exécution de toutes les instructions précédentes d'affectation et de test n'a eu jusqu'à maintenant aucun effet visible pour l'utilisateur.

- 160 END

Fin de l'exécution ; l'ordinateur est prêt pour une autre tâche : exécuter à nouveau le programme, ou en recevoir un autre, etc...

Le détail des instructions d'affectation est donné au chapitre 2, et celui des instructions d'entrée-sortie au chapitre 3.

Voici ce qu'on obtient lors d'une exécution réelle sur ordinateur :



§ 1-2 INTERPRETEUR - SYSTEME

Un ordinateur, en tant qu'ensemble de circuits électroniques (hardware), n'est capable d'exécuter que les programmes écrits dans son 'propre' langage, dit langage-machine. Les langages-machines varient suivant les constructeurs et les modèles, bien qu'ils aient tous, dans l'état actuel de la technologie, de nombreux traits en commun. Un langage-machine, même codé sous une forme 'lisible' par un humain (et non, comme dans l'ordinateur, par des groupes de bits[†]), est très fastidieux à employer, car les instructions qui constituent le langage-machine sont très élémentaires. Personne, sauf exception, n'utilise directement le langage-machine pour écrire un programme.

On utilise à la place des langages 'universels' de programmation, comme BASIC, dont les instructions sont beaucoup plus puissantes, et qui sont indépendants, à quelques détails près, de la machine utilisée.

Comment l'ordinateur peut-il exécuter un programme écrit dans un langage qui n'est pas le sien, BASIC par exemple ? Deux types de solutions sont possibles:

- un compilateur BASIC est un programme qui traduit tout programme BASIC (appelé en l'occurence programme - source) en langage - machine; le résultat de la compilation est appelé programme - objet. Pour exécuter un programme BASIC, l'ordinateur peut exécuter le programme - objet obtenu par compilation.
- un interpréteur BASIC est un programme qui exécute les instructions BASIC à mesure qu'il les analyse.

Compilateur ou interpréteur sont des programmes fournis usuellement par le constructeur, ou par des entreprises spécialisées; ils font partie du logiciel (software) de l'ordinateur. Ils résident dans des mémoires 'mortes' (ROM[††]), ou sont stockés sur des disques magnétiques que l'ordinateur 'lit' lorsque c'est nécessaire. Ils sont écrits en langage - machine. On passe d'ailleurs, pour les écrire, par l'intermédiaire d'un langage d'assemblage: un langage d'assemblage comporte les mêmes instructions que le langage - machine, mais codées de façon 'lisible', et assorties de facilités considérables d'utilisation: les registres de mémoire peuvent recevoir des noms, etc... Un programme écrit en langage

---

[†]Bit = Binary digit. Un bit est l'unité élémentaire d'information, susceptible de prendre deux valeurs, représentées par 0 et 1.

[††]ROM = Read Only Memory

d'assemblage est converti en langage - machine par un assembleur. Un assembleur est un programme relativement simple (la traduction langage d'assemblage →lan--gage - machine est beaucoup plus directe que la traduction BASIC →langage - machine), écrit, nécessité oblige, en langage - machine.

Un interpréteur est en général plus facile à construire et à utiliser qu'un compilateur; un interpréteur facilite la mise au point d'un programme BASIC, car on peut éventuellement en suivre l'interprétation instruction par instruction. Les ordinateurs portables sont en général équipés d'interpréteurs. En revanche un programme compilé est exécuté beaucoup plus rapidement par l'ordinateur, puisque le résultat de la compilation est un programme écrit en langage - machine.

L'utilisation d'un langage évolué comme BASIC n'offre pas que des avantages. Lorsqu'on désire écrire un programme vraiment efficace, c'est à dire économisant au mieux mémoire et/ou temps d'exécution, il est préférable d'utiliser le langage d'assemblage de l'ordinateur, qui seul permet un contrôle fin de l'exécution.

Système.

Le logiciel de l'ordinateur comprend aussi un programme qui supervise la bonne marche de l'ensemble des opérations: entrée de programmes, exécution, inter--ruptions (volontaires ou dûes à des erreurs), envoi de messages d'erreur, lecture ou écriture sur bande magnétique, etc.... Ce programme est appelé système d'ex--ploitation, ou plus brièvement système. Il travaille quand l'ordinateur semble ne rien faire, puisque c'est lui qui réagit à chacune des touches que l'utilisateur peut frapper au clavier, pour les analyser et décider de l'action a exécuter ; plus généralement le système gère les entrées-sorties : clavier, écran, bandes magnétiques, imprimantes. Le système devient très complexe sur les ordinateurs qui travaillent en temps partagé (cas où plusieurs utilisateurs ont accès simultanément à l'ordinateur).

L'utilisateur peut donner des instructions au système par l'intermédiaire de commandes. Les commandes ne doivent pas être confondues avec des instructions BASIC ; elles ne peuvent faire partie d'un programme, et ne portent pas de numéro ; elles dépendent du système, et ne font donc pas partie du langage BASIC. Les deux commandes principales, qu'on retrouve sur tous les systèmes, sont :

- RUN : cette commande lance l'exécution du programme situé à ce moment-là en mémoire centrale

- LIST : cette commande provoque l'affichage, ou l'impression, du programme.

D'autres commandes permettent en particulier de transférer un programme sur bande magnétique (c'est en général SAVE), ou inversement de rappeler en mémoire centrale un programme sur bande (LOAD).

Consultez le manuel de référence de votre système.

Editeur.

Le logiciel d'un ordinateur comprend aussi un éditeur de programmes, qui fournit à l'utilisateur des facilités pour entrer et modifier ses programmes BASIC. L'éditeur peut être plus ou moins développé; de façon générale:

- les instructions d'un programme BASIC ne sont pas exécutées dans leur ordre d'entrée par l'utilisateur, mais dans l'ordre de leurs numéros. En numérotant ses instructions de 10 en 10, et en commençant par le numéro 100, l'utilisateur a donc la possibilité d'introduire, quand il le désire, de nouvelles instructions dans son programme, à la place qu'il désire. Par exemple pour intercaler une instruction supplémentaire entre les instructions 160 et 170, il suffit de l'entrer avec le numéro 165; pour rajouter des instructions en début de programme, il suffit de les entrer avec des numéros inférieurs à 100.

- pour modifier une instruction d'un programme, il suffit d'entrer une nouvelle instruction portant même numéro: l'ancienne est alors automatiquement effacée. Sur les appareils munis d'un écran de visualisation, l'éditeur permet en outre, le plus souvent, de rappeler une instruction sur l'écran et de la modifier; c'est commode lorsque les modifications ne portent que sur des détails.

Consultez le manuel de référence de votre système pour connaitre ses facilités d'édition.

Précisons que lorsqu'on entre une instruction BASIC:

- elle est d'abord stockée dans des registres spéciaux, qui forment une mémoire - tampon.

- sa syntaxe est ensuite analysée, et l'instruction n'est acceptée et transférée en mémoire centrale que si cette syntaxe est correcte. Sinon un message d'erreur est envoyé à l'utilisateur, qui doit apporter les corrections nécessaires avant d'entrer à nouveau l'instruction.

Exercices

1) L'algorithme d'Euclide décrit dans la section 1.1 fonctionne-t-il correctement dans le cas : a < b ?

2) Que se passe-t-il à l'exécution du programme suivant (programme PGCD où les instructions 120 et 130 ont été interverties) ?

```
0100 INPUT A,B
0110 R=A-B*INT(A/B)
0120 B=R
0130 A=B
0140 IF R≠0 GOTO 0110
0150 PRINT A
0160 END
```

3) Soit le programme

```
0100 INPUT A,B
0110 IF B=0 GOTO 0160
0120 R=A-B*INT(A/B)
0130 A=B
0140 B=R
0150 GOTO 0110
0160 PRINT A
0170 END
```

Ecrire l'organigramme correspondant à ce programme, et vérifier qu'il constitue une description correcte de l'algorithme d'Euclide.

2 INSTRUCTIONS D'AFFECTATION

Les instructions d'affectation sont du type :

$$X \longleftarrow f(A, B, C...)$$

où A, B, C..., X sont des noms de variables, et f une fonction. Une telle instruction est toujours exécutée selon le schéma suivant :

- d'abord évaluation de f(A, B, C...) en fonction des valeurs actuelles de A, B, C...

- puis attribution de la valeur ainsi calculée à la variable X.

L'ordre des opérations est en effet important pour donner un sens à des instructions très courantes comme :

$$N \longleftarrow N+1$$

Cette instruction signifie : "Augmenter de 1 la valeur du registre N".

En BASIC le symbole d'affectation est = , ce qui donne l'instruction

$$N = N+1$$

qu'il ne faut évidemment pas prendre pour une égalité au sens habituel du terme, ou pour une équation.

Ce chapitre est divisé en trois sections :

- quels sont les noms de variables autorisés en BASIC ?

- comment exprimer les opérations algébriques élémentaires (somme, produit, etc ...) ?

- quelles sont les fonctions mathématiques usuelles disponibles en BASIC ?

§ 2-1 NOMS DE VARIABLES

Les variables destinées en BASIC à recevoir une valeur numérique scalaire sont désignées par une lettre, ou une lettre suivies d'un chiffre. Les noms autorisés sont

donc :

A, B, C..... Z

A0, A1, A2,.....A9, B0, B1..... Z9

Il n'y a pas de distinction, en BASIC, entre des variables destinées à recevoir des valeurs entières, et celles destinées à recevoir des valeurs décimales. Comme prix à payer pour cette simplification, les programmes d'arithmétique occupent souvent une place excessive en mémoire (un entier étant assimilé à un décimal ne comportant que des zéros après la "virgule") ; pire, certains interpréteurs peu scrupuleux trouvent comme résultat de la division de 9 par 3 : 2. 999998, ou quelque chose de ce genre, jugeant qu'une précision de 10^{-6} est suffisante. De là a en déduire que 9 n'est pas un multiple de 3, il n'y a qu'un pas... De tels interpréteurs sont heureusement rares.

D'autre part, nous rencontrerons d'autres types de variables par la suite : vecteurs et matrices (chapitre 5), chaînes de caractères (chapitre 7) ; nous verrons alors comment distinguer les noms de telles variables de ceux des variables numériques simples que nous considérons actuellement.

§ 2-2 EXPRESSIONS ALGEBRIQUES

Elles sont composées à partir de constantes, ou de noms de variables, et d'opérateurs arithmétiques, qui sont :

```
+   POUR L'ADDITION
-   POUR LA SOUSTRACTION
*   POUR LA MULTIPLICATION (ATTENTION:CE N'EST PAS x)
/   POUR LA DIVISION
↑ OU **  POUR L'ELEVATION A UNE PUISSANCE
```

Exemples

```
2*A+1        A+B        A+X*Y        A↑2-B↑2
```

Comment sont comprises ces formules par l'ordinateur ? Par exemple :

```
2*A+1        signifie-t-il   2a + 1
                        ou   2(a + 1) ?
```

Les conventions adoptées en BASIC sont aussi voisines que possible du langage mathématique usuel : lors du calcul de la valeur d'une expression, les multiplications sont effectuées avant les additions, et donc :

```
2*A+1        signifie  2a + 1
```

La traduction BASIC de 2(a + 1) utilise elle aussi des parenthèses :

```
2*(A+1)
```

Attention à ne pas oublier le signe * .

De façon précise les opérateurs sont classés suivant une hiérarchie :

```
NIVEAU 3: ↑ (EXPONENTIATION)
NIVEAU 2: *  ET  /
NIVEAU 1: +  ET  -
```

Lors de l'évaluation d'une expression, l'ordinateur effectue d'abord les opérations de niveau 3, puis celles de niveau 2, enfin celles de niveau 1 ; à niveau hiérarchique égal, les opérations sont exécutées de gauche à droite ; des parenthèses peuvent

modifier, suivant les règles usuelles, l'ordre des opérations.

Exemples :

Formules BASIC	Formules mathématiques ordinaires correspondant
A+X*Y	$a + xy$
A↑2-B↑2	$a^2 - b^2$
2↑N+1	$2^n + 1$
2↑(N+1)	2^{n+1}
A/B+C	$\frac{a}{b} + c$

[Attention : l'obligation d'écrire sur une seule ligne peut conduire ici à une erreur. La règle est : priorité de la division sur la somme].

A/(B+C)	$\frac{a}{b+c}$
A/B*C	$\frac{a}{b} \times c$

[A niveau égal dans la hiérarchie, les opérations sont effectuées de gauche à droite]

A/(B*C)	$\frac{a}{b \times c}$

Les règles ci-dessus permettent de traduire facilement en langage BASIC les formule mathématiques usuelles, à condition d'y porter un minimum d'attention. La section 2.3 de ce chapitre sera consacrée aux expressions numériques faisant intervenir des fonctio mathématiques classiques : logarithmes, exponentielle, sinus, etc...

Il est important de bien noter que le signe = dans une instruction du type :

(1) v = < expression numérique >

désigne une opération d'affectation ; une telle instruction ne doit pas être confondue avec une équation..

Exemples :

(i) X + 2 = A

n'est pas une instruction BASIC valide ; elle sera refusée par l'ordinateur qui enverra à l'utilisateur un message d'erreur.

(ii) N = N + 1

est au contraire une instruction BASIC valide, même courante ; lors de son exécution, la valeur de la variable N est incrémentée de 1.

Pour comprendre ce genre d'instructions, il suffit de se rappeler que l'expression numérique à droite du signe = est d'abord évaluée, et que le résultat est ensuite affecté à la variable située à gauche du signe =.

Quelques précisions sur les constantes :

- conformément à l'usage anglo-saxon, BASIC utilise un point décimal et non une virgule. Exemples :

0.5 3.14 -0.6931

- il existe aussi une notation avec mantisse et exposant :

3 E 15 est la traduction BASIC de 3×10^{15}

2.45 E-3 est la traduction BASIC de 2.45×10^{-3}

- la constante π est représentée en BASIC par un symbole particulier, variable selon les systèmes. C'est &PI sur IBM 5100.

§ 2-3 FONCTIONS INCORPOREES

Le calcul des fonctions mathématiques élémentaires : logarithme , exponentielle, sinus, cosinus, etc... est un problème intéressant ; mais l'utilisateur du langage BASIC n'a pas à s'en préoccuper, car ces fonctions sont incorporées au langage.

Par exemple, l'interpréteur "comprend" une expression telle que SIN(X), et déclenche le calcul de la valeur de sin **x** (où **x** désigne la valeur affectée à cet instant à X) au moyen d'un programme qui fait partie du logiciel de l'ordinateur.

Toutes ces fonctions incorporées ont un nom de trois lettres, et l'argument doit toujours être placé entre parenthèses. Exemples d'instructions BASIC valides :

```
Y=SIN(X)
A=LOG(2*B+1)
```

Les fonctions incorporées peuvent intervenir dans la formation d'expressions numériques complexes, pourvu que la syntaxe usuelle soit respectée :

calcul de $3a + \sin^2 x$: `Y=3*A+(SIN(X))↑2` (Le résultat est en outre affecté à Y)

calcul de $x^2 \sin \frac{1}{x}$: `V=X↑2*SIN(1/X)` (Le résultat est en outre affecté à V)

Attention : Ces instructions ont le sens informatique habituel : la valeur numérique de l'expression située à droite du signe = est évaluée, en utilisant les valeurs numériques contenues dans les registres A, X, etc..., et est affectée au registre dont le nom figure à gauche du signe =. Ces instructions ne permettent pas de définir une fonction, et d'utiliser par exemple, par la suite l'expression : Y(X) ; cette expression n'a pas de sens en BASIC. Nous verrons au chapitre 6 comment définir en BASIC des fonctions autres que les fonctions incorporées.

Remarque

Une instruction telle que : X = SQR(-1) (racine carrée de -1) provoque à l'exécution l'apparition d'un message d'erreur et l'interruption de l'exécution du programme.

Liste des principales fonctions incorporées en BASIC :

Nom	Fonction
ABS	Valeur absolue
INT	Partie entière (INT est l'abréviation de INTEGER).
SQR	Racine carrée (SQR est l'abréviation de SQUARE)
DEG	Conversion de radians en degrés
RAD	Conversion de degrés en radians
SIN	Sinus
COS	Cosinus
TAN	Tangente
ASN	Arcsinus
ACS	Arccosinus
ATN	Arctangente
EXP	Exponentielle
LOG	Logarithme népérien
LGT	Logarithme en base 10
LTW	Logarithme en base 2
RND	Fournit un nombre au hasard compris entre 0 et 1 (RND est l'abréviation de RANDOM)

Pour les fonctions trigonométriques, une commande spéciale du système (et qui varie avec l'ordinateur) permet parfois de préciser si l'on travaille en radians, degrés ou grades ; l'unité standard est le radian.

La fonction RND est surprenante : le programme correspondant du logiciel calcule la valeur y_1 d'une certaine fonction H pour un argument x stocké dans une mémoire spéciale, disons α, puis stocke y_1 dans cette mémoire α ; à la prochaine utilisation de la fonction RND, le résultat en sera donc : $y_2 = H(y_1)$

et ainsi de suite par itérations successives. La fonction H est choisie de telle sorte que la suite $y_1, y_2, \ldots, y_n, \ldots$ soit distribuée "au hasard" : sur cette notion délicate, et sur les fonctions H qui peuvent être utilisées, cf chapitre 9.

La syntaxe exacte de la fonction RND varie suivant les systèmes.

Pour terminer cette section, voici un petit programme qui convertit coordonnées polaires en coordonnées cartésiennes (l'angle polaire étant introduit en degrés) :

```
0100 INPUT R,A
0105 A=RAD(A)
0110 X=R*COS(A)
0120 Y=R*SIN(A)
0130 PRINT X,Y

RUN
?
1,45
 .707107             .707107

RUN
?
3,-30
 2.598076           -1.5


 READY
```

valeurs entrées par l'utilisateur

réponse de l'ordinateur

et voici le programme inverse (passage de cartésiennes en polaires) :

```
0100 INPUT X,Y

0110 R=SQR(X↑2+Y↑2)
0120 A=ACS(X/R)
0130 IF Y≥0 GOTO 0150
0140 A=-A
0150 PRINT R,DEG(A)

RUN
?
1,1
 1.414214           45.000000

RUN
?
2.598076,-1.5
 3.000000           -30.000002


 READY
```

Les lignes 130 et 140 permettent de changer le signe de A, lorsque c'est nécessaire.

Pour ce genre de calculs, il vaut évidemment mieux se contenter d'une calculatrice de poche. Mais si vous avez un ordinateur sous la main, vous pouvez tester votre bonne compréhension de ce chapitre en écrivant de petits programmes de ce genre, pour effectuer tous les calculs qui vous passent par la tête. Testez vous-mêmes la correction de vos programmes sur des valeurs d'essai pour lesquelles le calcul à la main est facile.

Note : Sur la plupart des systèmes l'instruction END est facultative. C'est pourquoi désormais nous l'omettons parfois.

3 ENTREES - SORTIES

Les entrées-sorties sont un mal nécessaire. Informatique signifie traitement de l'information : l'utilisateur entre des informations dans l'ordinateur pour que celui-ci les traite et sorte les nouvelles informations tirées de ce traitement ; par exemple l'utilisateur entre :

- un programme
- une série statistique de valeurs numériques,

et l'ordinateur sort la moyenne et l'écart type de cette statistique.

Dans ce chapitre, nous nous intéresserons aux instructions qui, à l'intérieur d'un programme, permettent l'entrée de données ou la sortie de résultats (et non à l'entrée de programmes eux-mêmes)

La difficulté des instructions d'entrée-sortie réside dans la grande variété des supports possibles d'informations (télétypes, bandes magnétiques, cartes perforées, imprimantes, etc...) et des codages possibles de cette information. Plus l'ordinateur est capable d'exploiter une grande variété d'informations, plus les instructions d'entrée-sortie sont complexes. En BASIC, celles-ci sont considérablement simplifiées, car BASIC est un langage d'initiation, et l'important est d'apprendre à traiter l'information ; en revanche BASIC, du moins sous sa forme standard, ne permet d'utiliser qu'un nombre restreint de matériels périphériques d'entrée ou de sortie des données. En pratique BASIC est d'ailleurs souvent implanté sur des micro-ordinateurs ne travaillant que sur un petit nombre de types d'information.

§ 3-1 INSTRUCTION INPUT

Cette instruction d'entrée, très fréquente, a déjà été rencontrée au chapitre 1. Sa forme générale est :

INPUT v_1 , v_2 , ... v_n

où les v_i sont des noms de variables.

Lorsque l'ordinateur, au cours de l'exécution d'un programme, rencontre cette instruction, il imprime un ? et l'exécution est interrompue : l'utilisateur doit alors fournir par l'intermédiaire de son clavier, les données demandées, sous forme de constantes séparées par des virgules ; une fois que l'utilisateur a appuyé sur la touche "retour de chariot" l'exécution du programme reprend. Les données fournies sont affectées dans l'ordre aux variables qui figurent dans l'instruction INPUT. Des exemples ont déjà été vus chapitre 1.

Si l'utilisateur ne fournit pas le nombre de données exigé par l'instruction INPUT, la réaction de l'ordinateur est variable suivant les systèmes : message d'erreur, ou impression d'un nouveau ?, etc...

Consultez le manuel de référence de votre système.

Caractéristiques de l'instruction INPUT :

- comme pour toutes les instructions d'entrée-sortie, attention aux détails d'écriture : les noms de variables qui suivent INPUT comme les réponses de l'utilisateur lors de l'exécution doivent être séparés par des virgules.

- c'est une instruction conversationnelle (c'est-à-dire qui permet une "conversation" entre l'utilisateur et l'ordinateur). Elle convient lorsque l'utilisateur a peu de données à fournir, et qu'il souhaite en changer facilement. La source des données est donc le clavier de l'utilisateur et leur format est celui des constantes en BASIC ; par exemple :

- 2, 3.14, 1.05E6, etc... †

A noter qu'en général le symbole utilisé pour représenter π n'est pas accepté comme réponse à un INPUT.

---

† Rappel 1.05 E 6 est la traduction en BASIC de 1.05×10^6

§ 3-2 INSTRUCTIONS READ et DATA

Les données sur lesquelles doit travailler un programme peuvent être, en BASIC, incorporées à celui-ci, au moyen de l'instruction DATA. Exemple :

```
200 DATA 2, 3, 5, 7, 11, 13, 17, 19
```

Les constantes, séparées par des virgules, qui suivent DATA, constituent une forme de fichier.

(Attention : pas de virgule après la dernière valeur.)

L'instruction READ sert à lire ce fichier ; sa forme générale est semblable à celle de INPUT :

READ v_1, v_2, ... , v_n

où les v_i sont des noms de variables

Voyons sur un exemple le fonctionnement d'une telle instruction :

```
100 READ A,B
110 X = A↑2+B↑2
120 PRINT A,B,X
130 GOTO 100
140 DATA 2,3,3,4,8,7
```

Simulons l'exécution du programme :

```
100 READ A,B
```

Les deux premières valeurs du fichier sont lues : la valeur 2 est affectée a A, et 3 à B. En même temps un pointeur marque dans le fichier la prochaine valeur à lire :

```
2 3 3 4 8 7
    ↑
```

```
110 X=A↑2+B↑2
120 PRINT A,B,X
```

On obtient l'impression suivante :

```
 2             3             13
```

```
130 GOTO 100
100 READ A,B
```

La lecture du fichier est reprise à partir de la position marquée par le pointeur : la valeur 3 est affectée à A, et 4 à B. Le pointeur est déplacé

```
2 3 3 4 8 7
      ↑
```

```
110 X=A↑2+B↑2
120 PRINT A,B,X
```

Impression :

```
3                4                25
```

```
130 GOTO 100
100 READ A,B
```

La valeur 8 est affectée à A, et 7 à B. Le pointeur est déplacé :

```
2 3 3 4 8 7
        ↑
```

```
110 X=A↑2+B↑2
120 PRINT A,B,X
```

Impression :

```
8                7                113
```

```
130 GOTO 100
100 READ A,B
```

Il n'y a plus de valeur à lire dans le fichier à partir de la position marquée par le pointeur ; un message d'erreur apparaît alors, et l'exécution du programme est terminée.

Exécution réelle sur ordinateur :

Les instructions DATA qui permettent de constituer un fichier peuvent être situées n'importe où dans le programme ; en pratique ou les regroupe en début ou en fin de programme.

Le programme précédent est équivalent au programme :

```
100 READ A,B
110 X=A↑2+B↑2
120 PRINT A,B,X
130 GOTO 100
140 DATA 2,3
150 DATA 3,4
160 DATA 8,7
```

La série des instructions DATA définit un <u>seul fichier</u>, dont les valeurs se suivent dans l'ordre des numéros des instructions DATA.

Certains systèmes possèdent une instruction qui permet de traiter le cas où il n'y a plus rien à lire dans le fichier : cette instruction peut être par exemple :

```
IF END THEN n
```

où n est un numéro d'instruction ; au cas où une instruction READ ne peut être exécutée parce que la fin du fichier est déjà atteinte, il n'y a pas interruption de l'exécution et apparition d'un message d'erreur, mais poursuite de l'exécution du programme à partir de l'instruction numéro n.

Consultez le manuel de référence de votre système pour voir s'il admet une instruction de ce type.

Une autre solution, valable sur tous les systèmes, est d'ajouter au fichier une valeur conventionnelle (ou, pour reprendre l'exemple précédent, un couple de valeurs)

qui sert de signal d'arrêt ; l'essentiel est que cette valeur conventionnelle d'arrêt ne risque pas d'être confondue avec une valeur normale du fichier.

Exemple :

```
100 READ A,B
105 IF A=0&B=0 GOTO 150
110 X=A↑2+B↑2
120 PRINT A,B,X
130 GOTO 100
140 DATA 2,3,3,4,8,7,0,0
150 END
```

l'instruction 105 signifie : si les valeurs de A et B sont nulles, aller en 150.

Instruction RESTORE

Cette instruction a pour effet de renvoyer le pointeur en début de fichier ; un programme peut ainsi effectuer plusieurs traitements successifs sur le même fichier.

Caractéristiques des instructions READ et DATA :

- la source de données est un fichier incorporé au programme. Un tel fichier est dit séquentiel : on ne peut accéder aux valeurs contenues dans le fichier que l'une après l'autre. C'est une différence fondamentale avec un vecteur (cf chapitre 5) : dans le cas d'un vecteur, on peut accéder directement, par exemple, au dixième élément, etc...

- le format des données est celui des constantes en BASIC

- pour modifier les données, il faut modifier les instructions DATA du programme (d'où la nécessité de les regrouper clairement)

- cette forme d'entrée de données convient si les données sont relativement nombreuses : les instructions READ et DATA sont utilisées surtout lorsque le programme traite des tableaux (cf chapitre 5) ; on évite alors d'avoir à entrer les données à chaque exécution du programme (comme c'est le cas avec des instructions INPUT), ce qui est particulièrement désagréable lors de la phase de mise au point du programme (correction des erreurs, améliorations, etc...)

Le choix entre INPUT et READ... DATA est parfois uniquement une question de goût.

§ 3-3 INSTRUCTION PRINT

L'utilisation de cette instruction est plus sophistiquée qu'on pourrait le croire. En effet, cette instruction n'indique pas seulement quoi imprimer, mais aussi où, et cette dernière caractéristique est précisée par des signes de ponctuation (virgules ou points-virgules) beaucoup moins inoffensifs que dans le langage courant. Quant au format des valeurs imprimées (nombre de décimales, ou choix, par exemple, entre 1000 et 1E3 †), il est déterminé automatiquement par le système et non par l'utilisateur ; si l'utilisateur veut agir sur ce format, il doit utiliser une autre instruction d'impression (cf section 3.4).

L'instruction

PRINT < expression >

où < expression > désigne une expression numérique (cf section 1.1), provoque l'impression de la valeur actuelle de cette expression, et le retour de la tête d'impression au début de la ligne suivante.

Par contre, lorsqu'une expression est suivie d'une virgule dans une instruction PRINT, une zone dite étendue de 16 au 18 caractères suivant les systèmes, est consacrée à l'impression de la valeur correspondante, et la tête d'impression reste positionnée au premier caractère suivant cette zone sur la ligne (sauf si on est arrivé en bout de ligne : il y a alors retour automatique au début de la ligne suivante).

---

† Rappel 1E3 est la traduction en BASIC de 10^3.

Exemple :

```
0100 INPUT X
0110 PRINT X,X↑2,X↑3
0120 PRINT 1/X,SQR(X)

RUN
?
3
 3                    9                    27
 .333333              1.732051

RUN
?
5
 5                    25                   125
 .2                   2.236068
```

Après exécution de l'instruction 110, la tête d'impression revient au début de la ligne suivante, parce que la dernière expression de l'instruction 110 (à savoir X↑3) n'est pas suivie d'une virgule. Par contre, si on rajoute cette virgule à la fin de l'instruction 110, on obtient :

```
0100 INPUT X
0110 PRINT X,X↑2,X↑3,
0120 PRINT 1/X,SQR(X)

RUN
?
3
 3                    9                    27                   .333333              1.732051

RUN
?
5
 5                    25                   125                  .2                   2.236068
```

Le programme est alors équivalent à :

```
100 INPUT X
110 PRINT X,X↑2,X↑3,1/X,SQR(X)
```

L'exemple suivant montre qu'on peut placer aussi une virgule juste après PRINT, ou utiliser des virgules consécutives, pour obtenir le "saut" d'une zone étendue :

```
0100 INPUT X
0110 PRINT X,X↑2,X↑3
0120 PRINT ,1/X,SQR(X)
0130 PRINT
0140 PRINT X,,X↑2,X↑3

RUN
?
3
 3                 9                 27
                   .333333           1.732051

 3                                   9                 27
```

L'instruction

```
130 PRINT
```

produit le déplacement de la tête d'impression au début de la ligne suivante : d'où ici un saut d'une ligne.

Utilisation du point virgule

Si, dans une instruction PRINT, une expression est suivie d'un point-virgule, celle-ci est imprimée, lors de l'exécution du programme, dans une zone condensée. La définition précise d'une zone condensée varie suivant les systèmes, mais garde toujours les caractéristiques suivantes :

- une zone condensée est plus courte qu'une zone étendue
- la longueur d'une zone condensée dépend de la valeur à imprimer

Exemple

```
0100 INPUT X
0110 PRINT X;X↑2;X↑3
0120 PRINT 1/X;SQR(X)

RUN
?
3
 3          9          27
 .333333    1.732051
RUN
?
1000
 1000       1000000       1000000000
 1E-3       31.622777
RUN
?
1.12345
 1.12345    1.262140      1.417951
 .890115    1.059929
```

Si on ajoute un point-virgule à la fin de l'instruction 110, on obtient

```
0100 INPUT X
0110 PRINT X;X↑2;X↑3;
0120 PRINT 1/X;SQR(X)

RUN
?
3
 3          9          27          .333333          1.732051
```

Le programme est alors équivalent à :

```
100 INPUT X
110 PRINT X;X↑2;X↑3;1/X;SQR(X)
```

Autres exemples :

```
 READY

0100 DATA 2,3,25,1.12345,0
0110 READ X
0115 IF X=0 GOTO 0140
0120 PRINT X;1/X;X↑2;X
0130 GOTO 0110
0140 END

RUN
 2        .5       4        2
 3        .333333  9        3
 25       4E-2     625      25
 1.12345  .890115  1.262140 1.12345
```

On voit que l'utilisation de zones condensées permet de gagner de la place, mais ne fournit pas en général un bon alignement en colonnes, contrairement à l'utilisation de zones étendues :

```
 READY                                                    28211

0100 DATA 2,3,25,1.12345,0
0110 READ X
0115 IF X=0 GOTO 0140
0120 PRINT X,1/X,X↑2,X
0130 GOTO 0110
0140 END

RUN
 2                 .5                4                 2
 3                 .333333          9                 3
 25                4E-2              625               25
 1.12345           .890115           1.262140          1.12345
```

On peut mêler, dans une même instruction, virgules et points-virgules : les valeurs suivies d'une virgule sont imprimées en zone étendue, celles suivies d'un point-virgule ssont imprimées en zone condensée.

Exemples :

```
0100 DATA 1,2.1234,1.E03
0110 READ A,B,C
0120 PRINT A,B;C

RUN
 1                    2.1234    1000
```

La valeur de A est imprimé en zone étendue, celle de B en zone condensée ; après impression de C, la tête d'impression revient au début de la ligne suivante

```
0100 DATA 1,2.1234,1.E03
0110 READ A,B,C
0120 PRINT A,B;C
0130 PRINT A;B

RUN
 1                    2.1234    1000
 1      2.1234
```

```
0100 DATA 1,2.1234,1.E03
0110 READ A,B,C
0120 PRINT A,B;C,
0130 PRINT A;B

RUN
 1                    2.1234    1000              1       2.1234
```

Dans le dernier exemple la valeur de A est imprimée une première fois en zone étendue ; puis B est imprimé en zone condensée, C en zone étendue, A en zone condensée ; enfin il y a retour à la ligne après la dernière impression de B.

Remarque

Si la place manque pour imprimer une zone, étendue ou condensée, parce que la tête d'impression arrive en fin de ligne, cette zone est automatiquement transféré en entier au début de la ligne suivante (les valeurs imprimées ne sont donc jamais coupées).

Impression de caractères

L'instruction PRINT permet aussi d'imprimer des textes ; ceux-ci doivent être placés entre apostrophes (simples ou doubles, cela dépend des systèmes), et sont alors imprimés tels quels (y compris les blancs). Un texte est aussi appelé chaîne de caractères.†

Si un texte entre apostrophes n'est suivi ni d'une virgule ni d'un point-virgule (et termine donc une instruction PRINT) il y a retour de la tête d'impression au début de la ligne suivante après impression du texte.

Si le texte est suivi d'une virgule, il est imprimé en zone étendue : 18 caractères, ou, si le texte est long : 36 caractères, 54, etc... (respectivement 16, 32, 48... sur d'autres systèmes).

Si le texte est suivi d'un point-virgule, il est imprimé en zone condensée : en général la longueur de la zone condensée est alors exactement égale à celle du texte. (c'est faux sur IBM 5100).

L'impression de caractères est très utile pour rendre agréable les sorties d'un programme. Reprenons par exemple le programme PGCD (section 1.1) et améliorons les sorties :

---

† Si la chaîne de caractères comprend elle-même une apostrophe, celle-ci doit être doublée

Exemple :

```
100 PRINT 'J''CAUSE BASIC'
RUN
J'CAUSE BASIC
```

```
0100 INPUT A,B
0110 PRINT 'LE PGCD DE';A;'ET';B;'VAUT.'
0120 R=A-B*INT(A/B)
0130 A=B
0140 B=R
0150 IF R≠0 GOTO 0120
0160 PRINT A
0170 END

RUN
?
195,55
LE PGCD DE 195  ET 55   VAUT:
 5

RUN
?
123456,654321
LE PGCD DE 123456  ET 654321  VAUT:
 3
```

Si on rajoute une virgule à la fin de l'instruction $_{110}$, on obtient :

```
0100 READ A,B
0105 IF A=0 GOTO 0200
0110 PRINT 'LE PGCD DE';A;'ET';B;'VAUT:',
0120 R=A-B*INT(A/B)
0130 A=B
0140 B=R
0150 IF R≠0 GOTO 0120
0160 PRINT A
0170 GOTO 0100
0180 DATA 195,55,123456,654321,12,1234,0,0
0200 END
```

```
RUN
LE PGCD DE 195   ET 55    VAUT:                5
LE PGCD DE 123456  ET 654321  VAUT:                 3
LE PGCD DE 12    ET 1234     VAUT:                2
```

Variante:

```
0110 PRINT 'LE PGCD DE';A,'ET';B,'VAUT:',

RUN
LE PGCD DE 195                ET 55                VAUT:                5
LE PGCD DE 123456             ET 654321            VAUT:                3
LE PGCD DE 12                 ET 1234              VAUT:                2
```

Attention

Si on remplace l'instruction 110 par :

```
110  PRINT 'LE PGCD DE A ET B VAUT :',
```

on obtient à l'exécution :

```
RUN
LE PGCD DE A ET B VAUT :        5
LE PGCD DE A ET B VAUT :        3
LE PGCD DE A ET B VAUT :        2
```

Cette fois A et B sont de simples caractères d'un texte, qui n'ont rien à voir avec les variables repérées par A et B, et l'impression obtenue est sans grand intérêt.

Note : rôle des "blancs" en BASIC

Pour un ordinateur, le caractère "blanc", obtenu en appuyant sur la barre d'espacement du clavier, est un caractère comme un autre.

BASIC, en règle générale, ne tient pas compte des blancs qui interviennent dans une instruction. Par exemple :

```
150 IF R≠0 GOTO 110
```

est équivalent à :

```
150 IF  R ≠ 0 GOTO   110
```

ou a :

```
150IFR≠0GOTO110
```

La dernière forme est seulement moins lisible pour l'utilisateur.

Il y a des exceptions de détail suivant les systèmes ; certains n'accepteront pas :

```
150 IF R≠0 GOTO 1 10
```

d'autres exigent un blanc après le numéro d'instruction etc...

Les blancs ne peuvent donc pas être utilisé$_s$, comme dans d'autres langages de programmation, pour séparer les expressions dans une instruction PRINT. De même, des instructions comme :

```
INPUT A B C
READ A B C
DATA 1 2 3 4
```

ne sont pas valides en BASIC, d'où l'importance des virgules comme signes de séparation à ne pas confondre surtout avec les points décimaux.

Le seul cas important où BASIC prend les blancs en considération est celui des chaînes de caractères, dont les blancs font partie au même titre que tout autre caractère.

Exercice

Que se passe-t-il à l'exécution si, par mégarde, on écrit le programme PGCD sous la forme :

```
0100 INPUT A,B
0110 R=A-B*INT(A/B)
0120 A=B
0130 B=R
0140 IF R≠0 GOTO 0110
0150 PRINT 'LE PGCD DE';A,'ET';B,'VAUT:';
0160 PRINT A
```

§ 3-4 INSTRUCTION PRINTUSING

Lors de l'exécution d'une instruction PRINT :

- le positionnement des valeurs à imprimer sur la ligne est contrôlé de façon assez grossière par l'utilisateur, au moyen des signes de ponctuation

- le format d'impression (notation en virgule fixe ou par mantisse et exposant, nombre de décimales) est déterminé entièrement par le système.

Si, pour une raison ou une autre, on souhaite un contrôle plus précis de l'impression, il faut utiliser l'instruction PRINTUSING.

Exemple :

```
0100 READ A,B
0110 IF A=0 GOTO 0210
0120 PRINT USING 0200,A,B;
0130 R=A-B*INT(A/B)
0140 A=B
0150 B=R
0160 IF R≠0 GOTO 0130
0170 PRINT A
0180 GOTO 0100
0190 DATA 195,55,123456,654321,12,1234,0,0
0200 :LE PGCD DE ####### ET ####### VAUT:
0210 END
```

```
RUN
LE PGCD DE     195 ET      55 VAUT: 5
LE PGCD DE  123456 ET  654321 VAUT: 3
LE PGCD DE      12 ET    1234 VAUT: 2
```

L'instruction 120 signifie : Imprimer en utilisant l'instruction 200, les valeurs de A et B (le point-virgule final signifie que l'impression ne doit pas être suivie d'un retour à la ligne).

L'instruction 200 est une instruction de format, nécessairement couplée à une instruction PRINTUSING. Une instruction de format commence par le symbole : (le symbole varie suivant les systèmes). Lors de l'exécution de l'instruction 120, la ligne 200 (ou plus exactement la partie de la ligne qui suit le symbole :) est imprimée telle quelle (y compris les blancs), à l'exception des symboles ######## qui sont remplacés par les valeurs de A et B.

La spécification ######## convient pour un entier de six chiffres au maximum (ou pour un entier négatif de cinq chiffres au maximum, car le signe - compte pour un caractère).

Si on remplace l'instruction 200 par :

```
200 :LE PGCD DE ##### ET ##### VAUT:
```

On obtient à l'exécution

```
RUN
LE PGCD DE  195 ET   55 VAUT : 5
LE PGCD DE **** ET **** VAUT : 3
LE PGCD DE   12 ET 1234 VAUT : 2
```

Les symboles **** marquent que la spécification ##### est inapte à représenter un entier de 6 chiffres.

Une spécification comme ####.## convient pour une valeur comprise entre -99.99 et 999.99 ; cette valeur sera imprimée avec exactement deux décimales.

Grâce aux instructions de format, on peut donc spécifier le nombre de décimales à imprimer, et obtenir un parfait alignement. Inversement, si on a mal calculé le nombre de chiffres avant la virgule, on risque des ennuis, ce qui n'arrive jamais avec une instruction PRINT.

Pour les autres spécifications utilisables dans une instruction de format, consultez votre manuel de référence.

§ 3-5 AUTRES SUPPORTS D'INFORMATION

Aussi bien pour l'entrée que pour la sortie d'informations, on utilise d'autres supports que le clavier (entrée) ou l'imprimante (sortie) ; en particulier on a souvent besoin de lire des données, ou au contraire d'écrire des résultats, sur une bande magnétique. Les instructions correspondantes, qui constituent une partie de ce qu'on appelle le traitement des fichiers, ne font pas partie du langage BASIC standard et sont donc variables suivant les systèmes : certains ont des instructions spéciales (GET, PUT, etc... sur IBM 5100), d'autres utilisent des variantes des instructions INPUT, PRINT, etc... Consultez votre manuel de référence.

De même pour les systèmes munis à la fois d'un écran et d'une imprimante, consultez votre manuel de référence pour savoir comment obtenir, à l'exécution d'une instruction PRINT, l'impression sur l'un ou l'autre de ces périphériques de sortie.

Sur IBM 5100, l'instruction PRINT provoque l'affichage sur écran ; l'impression sur papier est obtenu par l'instruction PRINT FLP,

Exemple : PRINT FLP, X,Y

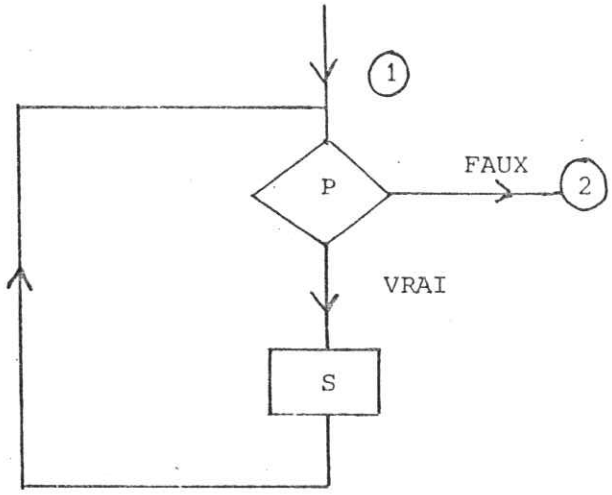
(F pour "fichier", LP pour "line Printer").

4 BOUCLES ET TESTS

§ 4-1 SCHEMAS PEPETITIFS

L'exécution répétitive d'une série d'instructions est au centre de presque tous les programmes. L'algorithme d'Euclide (cf chapitre 1) est par exemple un algorithme répétitif.

Lorsqu'on décrit un algorithme par un organigramme, la répétition d'une série d'instructions apparaît sous la forme d'une boucle dans le graphe. Le schéma fondamental est le suivant :



Organigramme 4.1

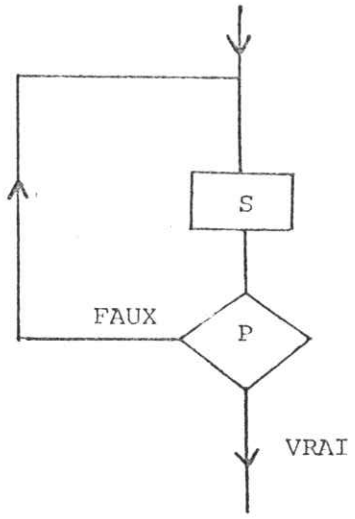
On entre dans la boucle au point (1), on teste si une certaine proposition P est vraie ou fausse ; tant que cette proposition reste vraie on effectue de façon répétitive la série d'instructions S ; lorsqu'elle devient fausse, on sort de la boucle par le point (2) (et on passe à l'exécution de la suite du programme).

P est à proprement parler un prédicat : sa valeur, VRAI ou FAUX, dépend de variables A, B, C ... ; à moins que le programme ne soit absurde, la série d'instructions S modifie les valeurs des variables A, B, C ... dont dépend P : ainsi la valeur de P peut, à un moment donné, basculer de VRAI à FAUX, et provoquer la sortie hors de la boucle.

On peut exprimer l'organigramme 4.1 par une seule instruction :

Tant que P faire S

Une variante fréquente est représenté par l'organigramme :



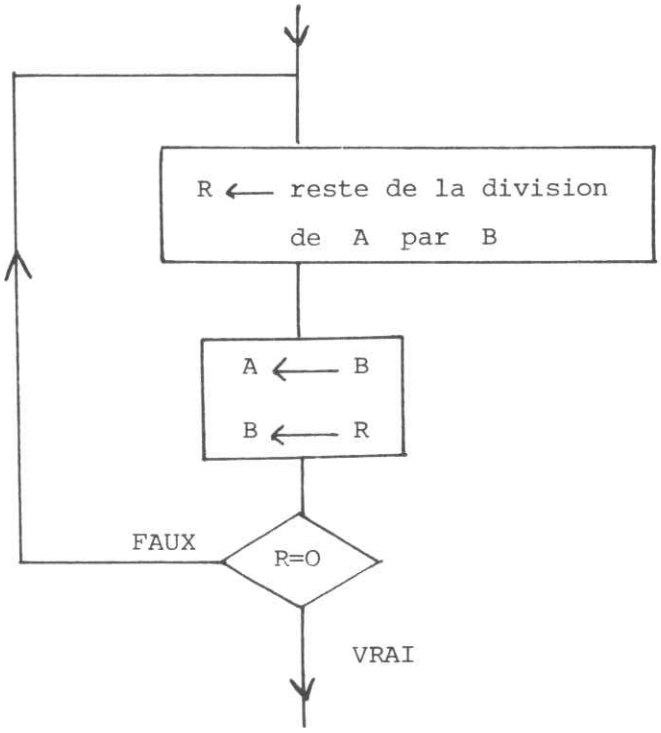
ou de façon équivalente, par l'instruction :

Répéter S jusqu'à P

La différence entre une boucle tant que ... faire et une boucle répéter ... jusqu'à tient donc seulement à la place du test de sortie de boucle : dans le premier cas, il est effectué avant exécution de la boucle ; dans le deuxième cas, après. Le plus souvent, le choix entre ces deux formules est arbitraire, bien que la première soit plus sûre.

Exemple 4.1.1

L'algorithme d'Euclide du chapitre 1 comporte essentiellement une boucle :



qu'on peut exprimer par :

```
Répéter  R ⟵ reste de la division de  A  par  B ;
         A ⟵ B ;   B ⟵ R
jusqu'à  R = O
```

L'algorithme complet peut donc être décrit simplement par :

```
Départ   A ⟵ a ;   B ⟵ b ;
Répéter  R ⟵ reste de la division de  A  par  B ;
         A ⟵ B ;  B ⟵ R
jusqu'à  R = O ;
z ⟵ A   Fin
```

Les points-virgules servent à séparer les instructions ; z désigne le résultat du calcul.

On peut aussi décrire l'algorithme par :

```
Départ    A ⟵ a ; B ⟵ b ;
Tant que  B ≠ 0  faire
     début
     R ⟵ reste de la division de  A  par  B ;
     A ⟵ B ;  B ⟵ R
     fin ;
z ⟵ A   Fin
```

Les mots début et fin servent de parenthèses pour isoler le bloc d'instructions à répéter.

Exemple 4.1.2

Soit la suite définie par la relation de récurrence :

$$U_{n+1} = \frac{1}{2}\left(U_n + \frac{a}{U_n}\right)$$

avec $U_o = 1$

où a est un réel † positif donné. Cette suite converge vers la racine carrée de a (cf chapitre 8). L'algorithme suivant, où ε désigne un réel positif donné fournit donc une valeur approchée de la racine carrée de a :

```
Départ    U ⟵ 1 ;
Répéter   U ⟵ 1/2 (U + a/U)
jusqu'à   |a - U²| < ε ;
z ⟵ U   Fin
```

† En fait l'ordinateur ne traite que des approximations décimales de réels.

BASIC ne comporte pas d'instruction correspondant à tant que ... faire et répéter... jusqu'à (sauf dans un cas particulier, que nous verrons section 4.2), et c'est dommage.

La traduction BASIC de tant que P faire S est du type (les numéros d'instruction sont arbitraires, et placés là à titre d'exemple):

Cette traduction s'effectue donc au moyen d'instructions de branchement :

- GOTO 300 est une instruction de branchement inconditionnel. La forme générale de ce type d'instruction est très simple.

GOTO n

où n est un numéro de ligne (si l'instruction numéro n ne figure pas dans le programme, un message d'erreur apparaîtra lors de l'exécution).

- IF... GOTO 500 est une instruction de branchement conditionnel. La forme générale de ce type d'instructions est :

IF < relation > GOTO n

ou, sur d'autres machines:

IF < relation > THEN n†

n désigne un numéro de ligne

< relation > désigne une relation entre deux expressions numériques (ou littérales ; cf chapitre 7).

---

† Nous parlons du BASIC standard, et non de certaines versions plus raffinées (qui ne sont d'ailleurs guère disponibles en France actuellement), où THEN peut être suivi d'une instruction, au lieu d'un numéro d'instruction.

Exemples :

```
IF A<B GOTO...
IF 2*N>A+B GOTO...
IF D≤SQR(N) GOTO...          (rappel : SQR(N) désigne √N)
IF A↑2+B↑2=C↑2 GOTO...
```

Les signes de relation peuvent être :

$<$, $\leqslant$, $=$, $>$, $\geqslant$, $\neq$

Sur les systèmes ne possédant pas le signe $\leqslant$, celui-ci est remplacé par : < = ; idem pour $\geqslant$; $\neq$ (lire : différent) est souvent remplacé par : < >

Certains systèmes permettent qu'une relation soit bâtie à partir de relations élémentaires en utilisant les opérations logiques :

ou : symboles employés suivant les systèmes : $\vee$, | , OR

et : symboles employés suivant les systèmes : $\wedge$, &, AND

non : symboles employés suivant les systèmes : $\sim$, $\neg$

Lors de l'exécution d'une instruction :

IF < relation > THEN n

ou IF < relation > GOTO n

la relation est évaluée par l'ordinateur : si elle est vérifiée à ce moment-là, il y a branchement à l'instruction numéro n ; sinon l'exécution du programme suit son cours normal (passage à l'instruction suivante).

De même, la traduction BASIC de :

Répéter S jusqu'à P est du type :

```
┌→ 300
│   ¦
│   ¦   (S)
│   ¦
│  480
│
└  490    IF ... GOTO 300
   500    Suite du programme
```

Remarque

Dans ce cas une seule instruction de branchement est nécessaire ; le prix à payer par contre est qu'une telle boucle est toujours exécutée au moins une fois, ce qui n'est pas forcément souhaitable : en principe un test avant la boucle est préférable à un test après.

Au chapitre 1 nous avons déjà vu un programme BASIC correspondant à l'algorithme d'Euclide. Voici le programme correspondant à l'exemple 4.1.2 :

Programme RACINE

```
0100 INPUT A,E
0110 U=1
0120 U=.5*(U+A/U)
0130 IF ABS(A-U↑2)>E GOTO 0120
0140 PRINT U

RUN
?
2,1E-5                    <-------  valeurs entrées par l'utilisateur
 1.414216                 <-------  réponse de l'ordinateur

RUN
?
3,1E-3
 1.732143
```

On prendra garde, en traduisant en BASIC des expressions comme :

Tant que P faire S

ou Répéter S jusqu'à P

que le prédicat qui intervient dans l'instruction IF ... GOTO est la négation de P.

D'autre part, pour rendre plus lisible la structure du programme BASIC, on peut utiliser une ligne de commentaire. En effet, en BASIC, toute ligne qui commence par REM est un commentaire ; une telle instruction n'est pas exécutée par l'ordinateur,

elle sert seulement à celui qui lit le programme.

Exemple :

Programme RACINE (bis)

```
0100 REM CALCUL APPROCHE DE LA RACINE CARREE DE A
0110 INPUT A,E
0120 U=1
0130 REM BOUCLE JUSQU'A APPROXIMATION < E
0140 U=.5*(U+A/U)
0150 IF ABS(A-U↑2)>E GOTO 0130
0160 PRINT U
0170 END
```

L'instruction 130 permet aussi, éventuellement, d'insérer des instructions en début de boucle sans modifier 150.

Poursuivons cette section par un autre exemple : la suite de Fibonacci.

Exemple 4.1.3

La suite de Fibonacci est définie par la relation de récurrence :

$$F_{n+1} = F_n + F_{n-1} \quad (n \geqslant 2)$$

avec $F_1 = F_2 = 1$

Elle possède nombre de propriétés mathématiques remarquables, et on la rencontre souvent là ou on l'attend la moins. Voici un programme qui calcule et imprime les valeurs de F_n inférieures à une valeur M donnée :

Programme FIBO 1

```
0100 INPUT M
0110 U=1
0120 V=1
0130 REM BOUCLE JUSQU'A V>M
0140 PRINT V;
0150 W=U+V
0160 U=V
0170 V=W
0180 IF V≤M GOTO 0130
0190 END

RUN
?
1000
 1      2      3      5      8      13     21     34     55     89     144
 233    377    610    987
```

Comme d'habitude, un petit nombre de registres de calcul (trois ici : U, V, W) suffisent pour obtenir tous les termes de la suite. Pour bien comprendre ce point, le lecteur pourra simuler l'exécution du programme, par exemple en entrant 15 comme réponse à l'instruction 100, puis en exécutant les instructions une à une, comme le fait l'ordinateur.

Remarques

1) L'impression en lignes "compactes" est dûe au signe ; de l'instruction 140.

Si on la remplace par :

```
140 PRINT V,
```

on obtient :

```
RUN
?
1000
 1              2              3              5
 8              13             21             34
 55             89             144            233
 377            610            987
```

(chaque ligne est divisée en quatre zones "étendues").

Si on écrit :

```
140 PRINT V
```

on obtient une impression en colonne :

```
RUN
?
200
 1
 2
 3
 5
 8
 13
 21
 34
 55
 89
 144
```

2) L'ordre des instructions

```
160 U=V
```

et `170 V=W`

est fondamental, comme le montre le tableau ci-dessous des affectations successives :

Programme correct :	Programme incorrect :
150 W=U+V	150 W=U+V
160 U=V	160 V=W
170 V=W	170 U=V

A l'exécution on obtient successivement :

Programme correct	Programme incorrect
U ← 1 , V ← 1	U ← 1 , V ← 1
W ← 2 , U ← 1 , V ← 2	W ← 2 , V ← 2 , U ← 2
W ← 3 , U ← 2 , V ← 3	W ← 4 , V ← 4 , U ← 4
W ← 5 , U ← 3 , V ← 5	W ← 8 , V ← 8 , U ← 8
W ← 8 , etc...	W ← 16 , etc...

Si on intervertit les lignes 160 et 170, on calcule, de façon tirée par les cheveux, les puissances successives de 2 !

Terminons cette section par un cas élémentaire, mais fréquent, d'emploi des instructions de branchement pour obtenir une répétition. Lorsqu'un programme commence par une instruction du type INPUT N, il est fréquent qu'on désire faire fonctionner le programme pour diverses valeurs de N ; dans ce cas, pour éviter d'entrer plusieurs commandes RUN successives, on peut remplacer l'instruction END par une instruction GOTO qui renvoie à l'instruction INPUT.

Exemple :

```
0100 INPUT N
0110 PRINT N↑3
0120 GOTO 0100
```

ou bien de façon générale

```
100 INPUT ...

500 GOTO 100
```

De tels programmes "bouclent" ; après chaque exécution, l'ordinateur demande une nouvelle valeur des variables, et seule une interruption manuelle permet de sortir de ce cycle infernal.

On emploie souvent en fait une forme plus raffinée.

Exemple :

```
0100 INPUT N
0105 IF N=0 GOTO 0130
0110 PRINT N↑3
0120 GOTO 0100
0130 END
```

Ainsi, en entrant la valeur 0, on provoque la fin de l'exécution du programme.

On peut même écrire, pour que l'utilisateur soit au courant de ce "truc" :

```
0100 PRINT 'ENTRER UNE VALEUR DE N'
0110 PRINT 'OU 0 POUR TERMINER L''EXECUTION DU PROGRAMME'
0120 INPUT N
0130 IF N=0 GOTO 0160
0140 PRINT N↑3
0150 GOTO 0100
0160 END

RUN
ENTRER UNE VALEUR DE N
OU 0 POUR TERMINER L'EXECUTION DU PROGRAMME
?
5
 125
ENTRER UNE VALEUR DE N
OU 0 POUR TERMINER L'EXECUTION DU PROGRAMME
?
1.01
 1.030301
ENTRER UNE VALEUR DE N
OU 0 POUR TERMINER L'EXECUTION DU PROGRAMME
?

 0
READY
```

La plupart des programmes que nous verrons par la suite peuvent être modifiés de la sorte.

---

† Remarquer le doublement de l'apostrophe à l'intérieur de la chaîne de caractères.

§ 4-2 BOUCLES FOR ... NEXT

Il existe un cas, qu'on rencontre fréquemment dans la pratique, pour lequel BASIC comporte un couple spécial d'instructions de répétition.

Exemple 4.2.1

Voici un programme qui calcule les termes F_1, F_2, ... F_n de la suite de Fibonacci, n étant un entier donné :

Programme FIBO 2

```
0200 INPUT N
0210 U=1
0220 V=1
0230 FOR K=3 TO N
0240 W=U+V
0250 PRINT K,W
0260 U=V
0270 V=W
0280 NEXT K
0290 END
```

```
GO 200,RUN
?
10
 3                2
 4                3
 5                5
 6                8
 7                13
 8                21
 9                34
 10               55
```

Cette commande permet, sur IBM 5100, de commencer l'exécution du programme à une ligne choisie (il se trouve qu'ici les lignes 100 à 190 sont occupées par le programme FIBO 1).

Valeur de n entrée par l'utilisateur.

Valeurs de F_k

Valeurs de k

La répétition est obtenue par le couple d'instructions :

```
230 FOR K=3 TO N
280 NEXT K
```

qui ont l'effet suivant : la série d'instructions situées entre l'instruction 230 et l'instruction 280 est exécutée de façon répétitive pour :

K = 3

puis

K = 4

K = 5 etc...

jusqu'à

K = N

Ensuite l'exécution se poursuit par l'instruction qui suit l'instruction 280 (ici c'est l'instruction END).

Exemple 4.2.2

Ce programme fournit la table des carrés et des cubes des dix premiers entiers :

```
0100 FOR K=1 TO 10
0110 PRINT K,K↑2,K↑3
0120 NEXT K
```

```
RUN
 1                1                1
 2                4                8
 3                9                27
 4                16               64
 5                25               125
 6                36               216
 7                49               343
 8                64               512
 9                81               729
 10               100              1000
```

Exemple 4.2.3 : calcul de séries

Ce programme calcule et imprime les sommes :

$$\sum_{1}^{n} \frac{1}{j} \quad \text{et} \quad \sum_{1}^{n} \frac{1}{j^2}$$

La valeur de n est fournie par l'utilisateur en réponse à une instruction INPUT

```
0100 INPUT N
0110 S1=0
0120 S2=0
0130 FOR J=1 TO N
0140 S1=S1+1/J
0150 S2=S2+1/(J*J)
0160 NEXT J
0170 PRINT S1,S2
0180 END


RUN
?
100
 5.187378          1.634984
```

Dans les instructions 130 à 160 on aurait pu aussi bien utiliser le nom de variable K à la place de J, ou n'importe quel autre nom de variable non utilisé par ailleurs. J est une variable muette, comme dans les écritures mathématiques :

$$\sum_{j=1}^{n} \frac{1}{J} \quad \text{et} \quad \sum_{j=1}^{n} \frac{1}{j^2}$$

Remarques

- les règles de hiérarchie des opérateurs (priorité de la division et du produit sur l'addition) font que les instructions 140 et 150 donnent bien les résultats attendus. Par contre l'instruction

S2 = S2+1/J*J

serait équivalente à :

S2 = S2+1

car à hiérarchie égale, les opérations sont effectuées de gauche à droite, et donc :

1/J*J est équivalent à : (1/J)*J

- d'autre part, une remarque sur les temps de calcul : un ordinateur calcule vite, mais il ne faut cependant pas surestimer cette rapidité de calcul. Un mini-ordinateur portable WANG 2200 S exécute le programme 4.2.3 pour N = 1000, en environ 30 secondes. Il n'est donc guère question de faire tourner ce programme avec N = 1 000 000 (le temps nécessaire serait d'environ 500 minutes, c'est-à-dire 8 heures!) Même si sur des ordinateurs plus importants, les temps de calcul sont facilement divisés par 10 ou 100, on voit qu'ils ne sont pas négligeables. Un autre facteur intervient d'ailleurs, dès que la taille des calculs devient considérable : l'accumulation des erreurs d'arrondi. Par nécessité, les calculs sont effectués avec un nombre limité de chiffres significatifs, d'où à chaque fois une erreur d'arrondi ; l'accumulation de ces erreurs diminue progressivement la valeur du résultat fourni par l'ordinateur.

Il est très fréquent qu'on ait, comme dans l'exemple ci-dessus, à initialiser les valeurs de certaines variables avant l'exécution de la boucle (instructions 110 et 120 ici). Cette initialisation doit être évidemment assurée par des instructions situées avant l'instruction FOR ... TO. Voici un exemple où la variable doit recevoir 1 comme valeur initiale (et non 0 comme ci-dessus).

Exemple 4.2.4 : calcul d'une factorielle

Ce programme calcule et imprime la valeur de : n ! = n(n-1) (n-2) ... 3×2×1

La valeur de n est fournie par l'utilisateur, en réponse à une instruction INPUT.

Programme FACT

```
0100 INPUT N
0110 F=1
0120 FOR P=2 TO N
0130 F=F*P
0140 NEXT P
0150 PRINT 'FACTORIELLE ';N;' = ';F
0160 END

RUN
?
5
FACTORIELLE  5      =  120
RUN
?
10
FACTORIELLE  10     =  3628800
RUN
?
35
FACTORIELLE  35     =  1.033315E40
```

L'instruction 150 combine l'impression des contenus des variables numériques N et F avec des impressions de caractères. Rappelons qu'il suffit de placer ces caractères entre apostrophes pour qu'ils soient imprimés tels quels. On a laissé des blancs à l'intérieur de ces chaînes de caractères pour aérer l'impression (entre deux apostrophes, un blanc est un caractère comme un autre) ; c'est plus ou moins nécessaire

suivant les systèmes : tous ne traitent pas rigoureusement de la même façon le symbole ; dans une instruction PRINT.

Rappelons aussi qu'on peut, comme d'habitude, améliorer légèrement ce genre de programme en ajoutant les instructions suivantes :

```
105 IF N=0 GOTO 170
160 GOTO 100
170 END
```

Il est alors inutile de relancer chaque fois le programme par RUN ; l'instruction 105 permet d'échapper au cycle des INPUT en entrant la valeur 0 pour terminer l'exécution.

Si l'on oublie d'initialiser une boucle (ici : instruction 110), le résultat varie suivant l'ordinateur employé :

- sur certains systèmes (par exemple : WANG 2200 S, IBM 5100), la commande RUN a pour effet, entre autres, d'attribuer la valeur 0 à toutes les variables numériques du programme avant l'exécution proprement dite du programme ; dans ce cas tout se passera bien si, comme à l'exemple 4.2.3 cette initialisation automatique est celle qui convient ; mais dans le cas du programme FACT (où l'instruction 110 F=1 aurait été oubliée), on trouvera comme résultat F=0 ! (puisque l'instruction 130 fournira chaque fois F = 0 × P = 0)

- sur d'autres systèmes (par exemple, Hewlett Packard 9830, Tektronix 4051) -et cette solution est préférable logiquement-, la commande RUN ne modifie pas les valeurs attribuées aux variables lors de l'exécution de programmes antérieurs. Oublier d'initialiser une variable peut donc conduire à des résultats étranges, si cette variable a été utilisée précédemment ; si cette variable n'a jamais reçu de valeur auparavant, un message d'erreur apparaîtra (à l'instruction 130 du programme FACT, puisque le premier calcul de F * P est impossible, F n'ayant pas reçu de valeur).

En règle générale, il est recommandé de toujours procéder effectivement aux initialisations nécessaires, sans compter sur le système pour faire certaines initialisations à zéro.

La variable qui intervient dans les instructions FOR ... TO et NEXT n'est pas forcément utilisée dans les autres instructions. Voici un exemple :

Exemple 4.2.5

Ce programme réalise une simulation de n parties de pile ou face ; n est entré par l'utilisateur en réponse à une instruction INPUT.

Pour cela, on utilise la fonction RND, qui génère une suite de nombres "au hasard" (cf section 2.3), compris entre 0 et 1.

La fonction RND doit, sur IBM 5100, être utilisée sans argument. La probabilité pour qu'on ait : RND < 0.5 est donc de 1/2. D'où le programme (on convient qu'on a obtenu "face" si RND < 0.5, et "pile" sinon)

Programme PILE OU FACE

```
0100 INPUT N
0110 P=0
0120 FOR K=1 TO N
0130 IF RND<.5 GOTO 0150
0140 P=P+1
0150 NEXT K
0160 PRINT 'NOMBRE DE PILE: ';P
0170 PRINT 'FREQUENCE: ';P/N
0180 END
```

L'instruction 110 P=0 réalise l'initialisation de la boucle.

```
RUN
?
100
NOMBRE DE PILE:  46
FREQUENCE:  .46

RUN
?
300
NOMBRE DE PILE:  146
FREQUENCE:  .486667
```

La variable P compte le nombre de "pile". La variable K n'intervient que dans les instructions 120 et 150, pour assurer que la "pièce" est "lancée" n fois.

Forme générale de l'instruction FOR ... TO

Les formes d'instruction FOR ... TO rencontrées jusqu'ici sont très fréquentes, mais la forme générale de l'instruction est :

FOR X=a TO b [STEP c]

[Les crochets ne font pas partie de l'instruction. Ils indiquent que STEP c peut être omis.]

où : X désigne un nom de variable

a, b, c désignent des expressions numériques.

a et b constituent les bornes de variation de X, et c le pas de variation.

Exemples :

```
FOR K=1 TO 2*N+1 STEP 2
FOR A1=10 TO 1 STEP -1
FOR J=U TO V STEP (V-U)/10
```

Le corps de la boucle est alors exécuté pour les valeurs suivantes de X :

a, a+c, a+2c, a+3c etc...

tant qu'on a : $X \leqslant b$ (si le pas c est positif)

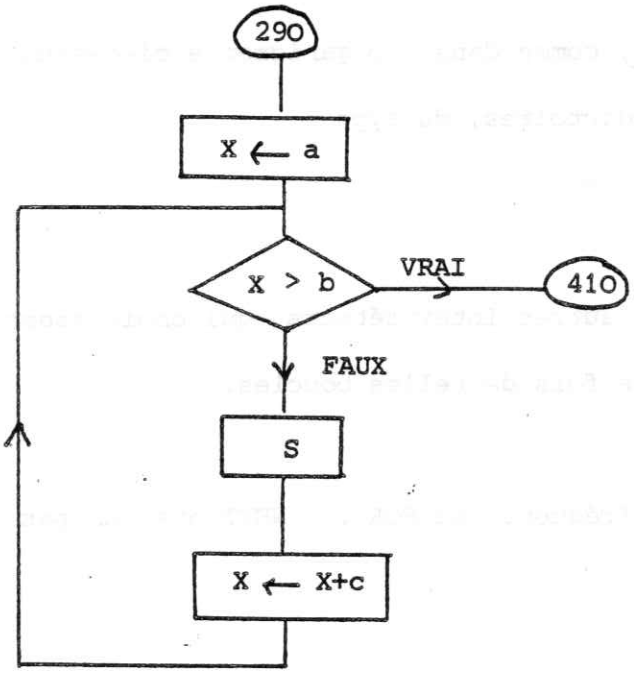
ou $X \geqslant b$ (si le pas c est négatif)

Si la partie STEP c de l'instruction est omise, comme précédemment, le pas c est alors égal, par défaut, à 1.

Une partie de programme du type :

```
300     FOR  X = a  TO b  STEP c
310 |
    |       (S)
390 |
400     NEXT X
```

correspond à l'organigramme suivant, si c est positif :



N.B. On suppose toutes les instructions numé-rotées de 10 en 10.

qu'on exprime par :

Pour X = a jusqu'à b pas c faire S

Remarque

La façon dont un ordinateur interprète la portion de programme ci-dessus :

```
300   FOR X = a   TO ...
  :
400   NEXT X
```

dépend en fait, dans les détails, de l'interpréteur BASIC et correspond rarement exactement au schéma précédent ; les variations portent sur trois points :

- a, b, c sont calculés une fois et une seule par l'interpréteur BASIC, qui ne tient pas compte, pour le contrôle de la boucle, de modifications apportées

éventuellement par l'exécution de S.

- valeur finale de X , après sortie de la boucle. Par exemple pour une boucle :

```
FOR  X = 1  TO 10
 |
 |
 |
NEXT X
```

la valeur de X en sortie de boucle est en général 10 (alors que d'après l'organigramme ci-dessus elle vaut 11).

- certains interpréteurs (ceux qui interprètent la boucle FOR ... NEXT comme une instruction tant que ... faire, comme dans l'organigramme ci-dessus) n'exécutent pas les boucles logiquement contradictoires, du type :

FOR X = a TO b STEP c

avec a < b et c < 0

ou bien a > b et c > 0 ; d'autres interpréteurs -qui choisissent la version répéter ... jusqu'à-, exécutent une fois de telles boucles.

Terminons sur une utilisation fréquente de FOR ... NEXT avec un pas différent de 1 :

Exemple 4.2.6

Il est souvent intéressant d'avoir un tableau de valeurs d'une fonction. Le programme suivant par exemple, permet de tabuler la fonction sinus. Les valeurs minimum et maximum de x, ainsi que le pas, sont à fournir par l'utilisateur, en réponse à une instruction INPUT (en degrés) :

Programme TAB

```
0100 PRINT 'X MINI, X MAXI, PAS';
0110 INPUT A,B,C
0120 FOR X=A TO B STEP C
0130 Y=SIN(RAD(X))
0140 PRINT X,Y
0150 NEXT X
0160 END
```

l'instruction 100 permet à l'utilisateur, lors de l'exécution du programme, de ne pas se tromper sur la nature et l'ordre des valeurs à entrer en réponse à l'instruction INPUT.

On peut évidemment modifier la ligne 130, et la remplacer par une ou plusieurs autres lignes, pour tabuler d'autres fonctions. On peut aussi tabuler plusieurs fonctions à la fois.

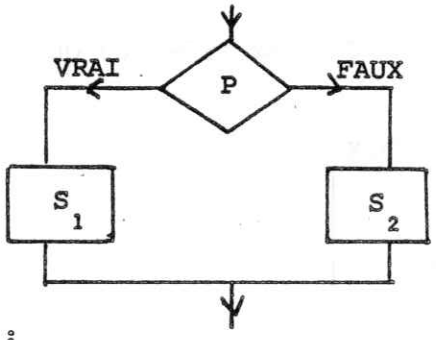
Si le pas est choisi petit, on peut voir si la fonction semble ou non continue, ou dérivable, et calculer la dérivée avec une assez bonne approximation.

Exemple de fonctionnement du programme précédent (on a retenu une précision de 6 décimales) :

```
RUN
X MINI, X MAXI, PAS
?
0,90,15
 0                    0
 15                   .258819
 30                   .500000
 45                   .707107
 60                   .866025
 75                   .965926
 90                   1
```

§ 4-3 SCHEMAS ALTERNATIFS

A côté des schémas répétitifs, il existe un deuxième type fondamental de schéma de programme, le schéma alternatif :



qu'on exprime par :

SI P alors S_1 sinon S_2.

La traduction BASIC sera un programme du type :

```
300   IF ... GOTO 400
310   |
      |
      |   (S1)
      |
380   |
390   GOTO 500
400   |
      |
      |   (S2)
      |
490   |
500   Suite du programme
```

Attention :

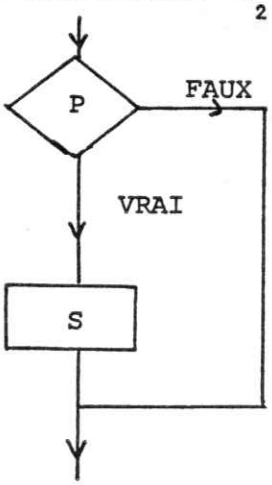
1) le prédicat qui intervient dans l'instruction 300 IF ... GOTO 400 est la négation du prédicat P de l'organigramme précédent.

2) ne pas oublier l'instruction 390 ! (sinon S_2 sera exécuté dans tous les cas).

Ce type de construction BASIC n'est guère élégant ; surtout la structure est peu lisible, d'autant plus que les instructions de branchement servent aussi, en BASIC, à réaliser les boucles qu'on ne peut exprimer par FOR ... NEXT.

Enfin, de branchement en branchement, on finit par se perdre. Ces difficultés sont hélas inhérentes au langage BASIC, du moins sous sa forme standard .

Dans certains cas, le bloc d'instruction S_2 est vide, ce qui donne l'organigramme :



qu'on exprime par :

Si P alors S

La traduction BASIC est plus simple (attention encore à employer la négation de P dans l'instruction IF) :

```
300   IF .... GOTO 400
310 |
 !  |  (S)
 !  |
390 |
400   Suite du programme
```

Alors qu'un programme simple se réduit souvent à une boucle, il se réduit rarement à un schéma alternatif. Le lecteur pourra bien entendu s'exercer cependant à programmer le traitement d'une équation du 2è degré ; mais en règle générale les schémas alternatifs apparaissent emboîtés avec des schémas répétitifs. Nous allons traiter quelques programmes d'arithmétique à titre d'exemples.

§ 4-3-1 DECOMPOSITION EN FACTEURS PREMIERS

On cherche à décomposer en facteurs premiers un entier positif n. Le principe est de chercher le plus petit diviseur (distinct de 1) d de n (d est donc premier), puis de décomposer n/d, etc... Plus précisément, la procédure peut s'énoncer [on désigne par z le vecteur résultat, c'est-à-dire le vecteur des facteurs premiers de n ; (z, D) désigne le vecteur z augmenté (par concaténation) de D†] :

```
Départ  N ← n ; z ← vecteur vide ;
Tant que  'N non premier" faire
     début
     D ← 'plus petit diviseur de  N' ;
     z ← z,D ;
     N ← N/D
     fin ;
z ← z,N
Fin
```

† Par exemple si z = (3, 5) et D = 11, (z, D) = (3,5,11)

Consulter la section 4.4 pour une description précise du langage utilisée :

Le point crucial est d'exprimer la condition 'N premier' ; or la recherche du plus petit diviseur D de N fournit en même temps ce critère : car lorsque dans cette recherche D^2 devient supérieur à N, on sait que N est premier. Sinon :

$$N = DD'$$

et comme D est le plus petit diviseur :

$$D^2 \leqslant DD' = N \text{ et contradiction}$$

D'où le nouvel algorithme :

```
Départ   N ⟵ n ; z ⟵ vecteur vide ; D ⟵ 2 ;

Tant que  D² ⩽ N  faire
    Si  D  divise  N
        alors   début
                z ⟵ z, D ;
                N ⟵ N/D  ; D ⟵ 2
                fin
        sinon   D ⟵ D+1 ;
z ⟵ z, N
Fin
```

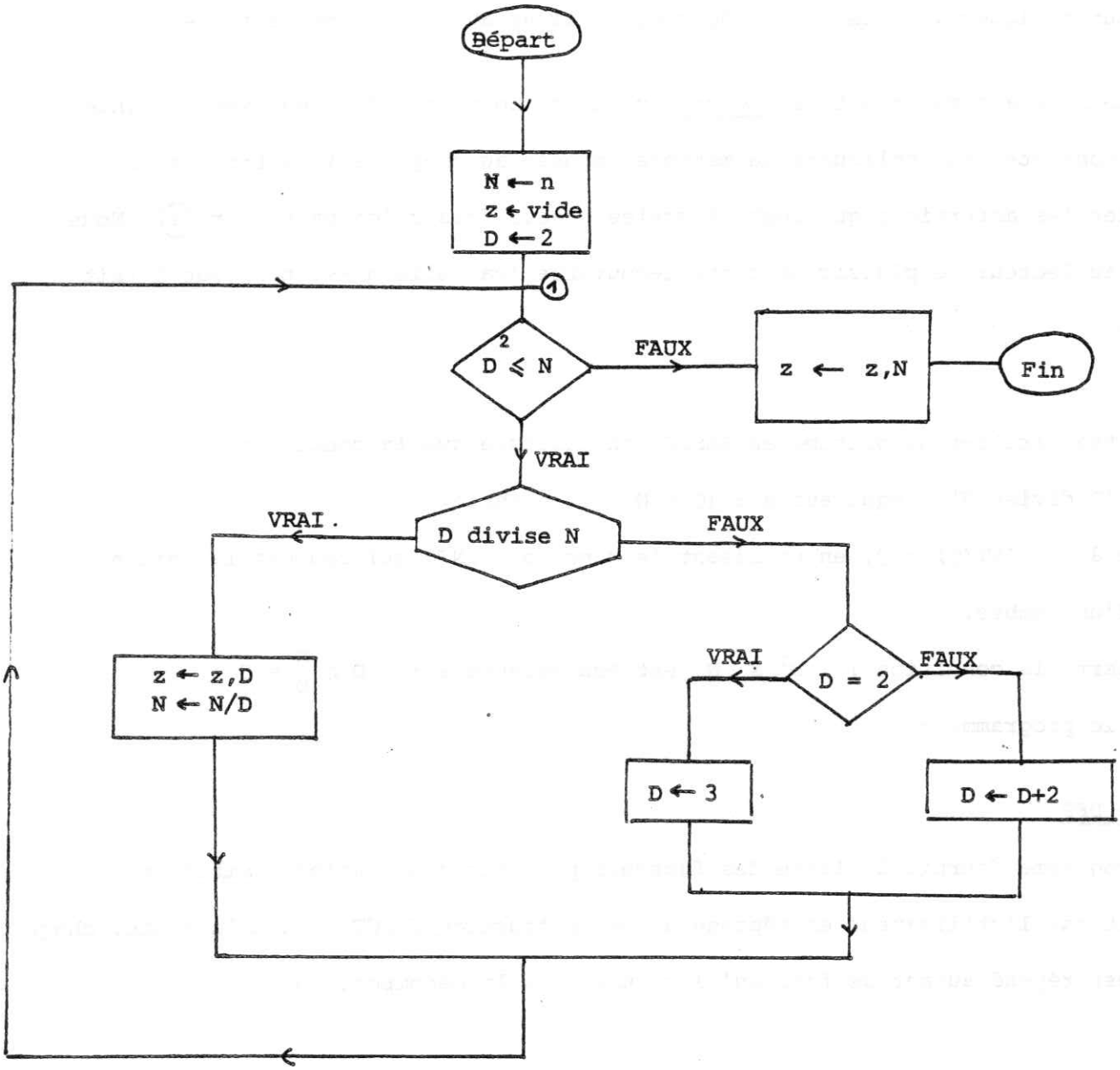
Remarque

Cet algorithme peut être amélioré en remplaçant l'instruction : D ⟵ D+1 par :

```
si  D=2   alors   D ⟵ 3
          sinon   D ⟵ D+2
```

Surtout un peu de réflexion montre que l'instruction D ⟵ 2 qui suit N ⟵ N/D est inutile, car N/D ne peut pas avoir de diviseur inférieur à D (qui est le plus petit diviseur de N).

L'organigramme correspondant est :



Organigramme 4.3.1

Rappel: z désigne le vecteur résultat, et z,D désigne le vecteur obtenu par concaténation de z et D.

On peut avoir de bonnes raisons de penser que cet algorithme est correct en le testant sur quelques valeurs de n. Nous conseillons au lecteur de le faire.

Le mieux serait cependant de prouver qu'il est correct. C'est un exercice intéressant qui consiste, en appliquant la méthode exposée au chapitre 1, à trouver en particulier les assertions qui restent vraies chaque fois qu'on passe par (1). Nous laissons au lecteur le plaisir de cette découverte (car elle n'est pas tout à fait immédiate).

Pour traduire cet algorithme en BASIC, on remarque que la condition :

(D divise N) équivaut à : (Q = N/D est entier)

ou encore à : INT(Q) = Q, en utilisant la fonction INT, qui fournit la partie entière d'un nombre.

D'autre part, la condition : $D^2 \leqslant N$ est équivalente à : $D \leqslant \frac{N}{D} = Q$

D'où le programme :

Programme DFP

Ce programme fournit la liste des facteurs premiers d'un entier positif n (introduit par l'utilisateur en réponse à une instruction INPUT). Dans la liste, chaque facteur est répété autant de fois qu'il figure dans la décomposition.

Les sorties anormales de boucles sont autorisées, en BASIC, mais sur certains systèmes, des sorties anormales répétées (par exemple une centaine de fois) peuvent provoquer des difficultés, dans la mesure où le système garde en mémoire, à chaque sortie anormale, qu'une boucle est "en cours" (car il ne peut pas savoir si la sortie est définitive ou non). Il peut y avoir alors surcharge de la zone de mémoire qui gère les boucles.

Ecrivons maintenant un programme qui fournisse une liste de nombres premiers.

Programme PRIME 2

L'utilisateur entre deux entiers positifs a, b avec $a < b$, en réponse à une instruction INPUT. Le programme fournit alors la liste des nombres premiers compris entre a et b.

On supposera, d'abord, pour se consacrer à l'essentiel, a impair

Le programme PRIME 2 peut alors s'écrire :

```
Départ   A ← a ;  B ← b ;
Pour  N = A jusqu'à  B  pas  2  faire
      Si  N  premier  alors  imprimer N ;
Fin
```

Il faut maintenant "raffiner", c'est-à-dire décrire précisément, le test "N premier ?" ; or cette question a déjà été traitée (programme PRIME 1)

En effet, partons du "squelette" du programme PRIME 1, à savoir :

PRIME

```
0130 FOR D=3 TO SQR(N) STEP 2
0140 Q=N/D
0150 IF INT(Q)=Q GOTO 0190
0160 NEXT D
```

Cette série d'instructions, que nous appellerons PRIME, a l'effet suivant, si N est impair :

- sortie en 170 si N est premier
- sortie en 190 sinon.

D'où le programme PRIME 2 :

```
0100 INPUT A,B
0110 FOR N=A TO B STEP 2
0130 FOR D=3 TO SQR(N) STEP 2
0140 Q=N/D
0150 IF INT(Q)=Q GOTO 0190
0160 NEXT D
0170 PRINT N;
0190 NEXT N
0200 END
```

Rappelons que ce programme, qui ne teste que les nombres impairs pour savoir s'ils sont premiers (instruction 110) fonctionnera correctement dans le seul cas a impair. Ceci peut être corrigé facilement en rajoutant les instructions 105 et 106 qui, dans le cas où a est pair, le remplacent par a+1 :

```
105 IF INT(A/2)≠A/2 THEN 110
106 A=A+1
```

Programme DFP:

```
0100 REM DECOMPOSITION EN FACTEURS PREMIERS
0110 INPUT N
0120 IF N=0 GOTO 0270
0130 C=1
0140 D=2
0150 REM BOUCLE TANT QUE D↑2 ≤ N
0160 Q=N/D
0170 IF Q<D GOTO 0250
0180 IF INT(Q)=Q GOTO 0220
0190 C=C+2
0200 D=C
0210 GOTO 0150
0220 PRINT D;
0230 N=Q
0240 GOTO 0150
0250 PRINT N
0260 GOTO 0110
0270 END
```

```
RUN
?
12
 2     2     3

?
13
 13

?
605
 5     11    11

?

123456
 2     2     2     2     2     2     3     643

?
0
```

L'introduction de la variable auxiliaire C (instructions 130, 190, 200) est une astuce qui permet de générer, sans test, la suite 2, 3, 5, 7, 9...

Le signe ; dans l'instruction 220 PRINT D ; permet l'impression en ligne des facteurs premiers.

Remarque

Le programme BASIC diffère légèrement de l'algorithme qui le précède, en ce sens que les facteurs successifs sont envoyés à l'imprimante, au lieu d'être concaténés à l'intérieur d'un vecteur résultat z.

Nous avons tenu cependant à donner cette forme à l'algorithme, car il est important que le résultat d'un algorithme formel soit clairement désigné ; sinon par exemple cela n'a plus guère de sens de prouver sa correction.

Etudions maintenant des programmes susceptibles de fournir des listes de nombres premiers.

§ 4-3-2 NOMBRES PREMIERS

Voici d'abord un programme qui teste si un entier positif donné n est premier.

Programme PRIME 1 †

```
0100 INPUT N
0110 D=2
0120 IF INT(N/2)=N/2 GOTO 0190
0130 FOR D=3 TO SQR(N) STEP 2
0140 Q=N/D
0150 IF INT(Q)=Q GOTO 0190
0160 NEXT D
0170 PRINT N;' EST PREMIER'
0180 STOP
0190 PRINT N;' EST DIVISIBLE PAR ';D
```

```
RUN
?
17
 17     EST PREMIER

RUN
?
63
 63     EST DIVISIBLE PAR  3
```

† Le mot anglais PRIME a l'avantage de désigner à lui seul un nombre premier...

On peut remplacer les instructions 140 et 150 par l'instruction unique :

```
150 IF INT(N/D)=N/D THEN 200
```

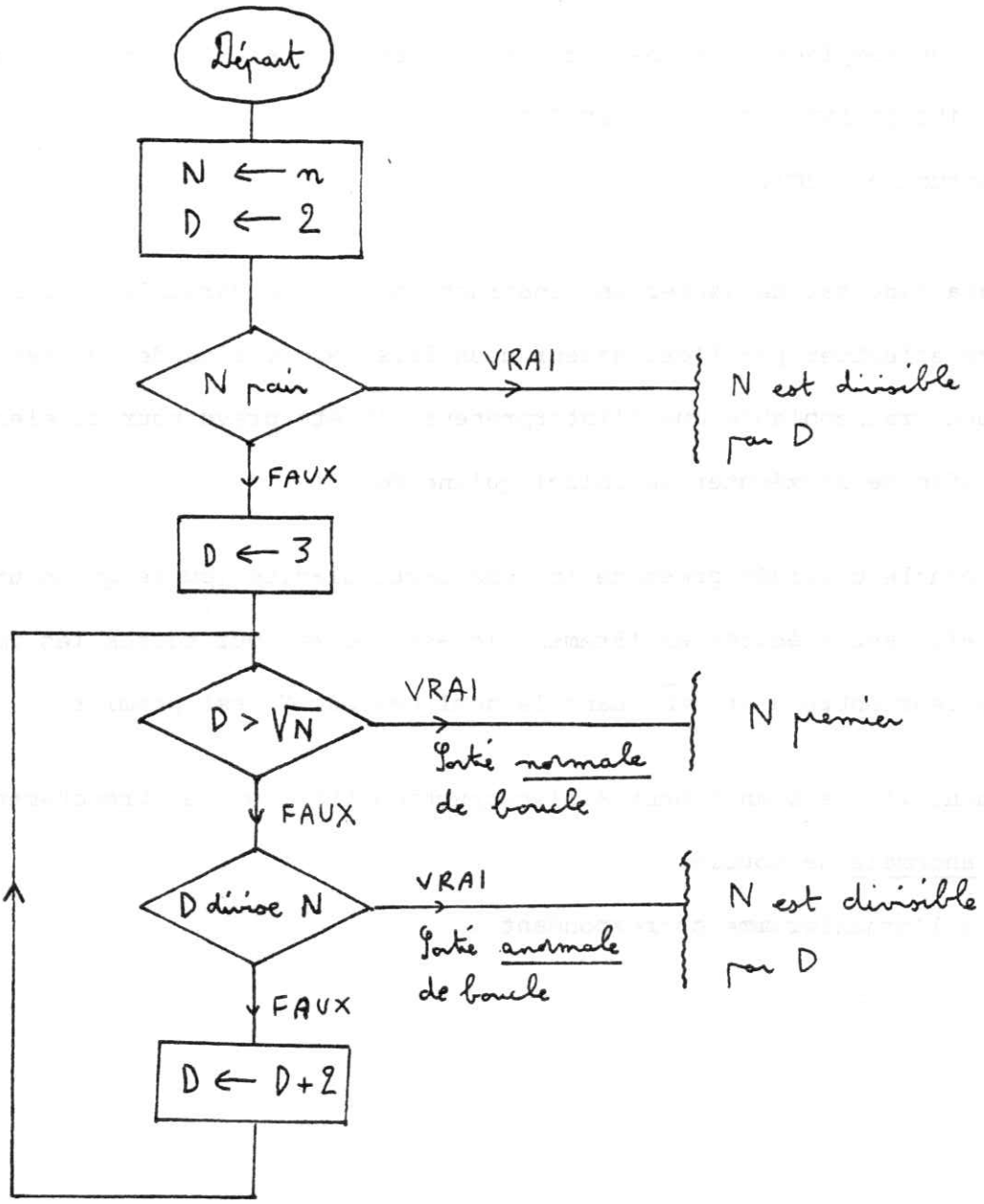
(cf instruction 120).

L'avantage est de gagner une instruction, et une variable (Q). L'inconvénient est de faire effectuer par l'ordinateur deux fois la division de N par D ; il est en effet peu vraisemblable que l'interpréteur ait été prévu pour déceler la répétition de N/D afin de n'exécuter le calcul qu'une fois !

La boucle utilisée présente ici une particularité (outre qu'on utilise un pas égal à 2) : elle est exécutée entièrement (c'est-à-dire pour toutes les valeurs impaires de D comprises entre 3 et $\sqrt{N}$) dans le seul cas où N est premier.

Sinon, il y a branchement à l'instruction 190 ; un tel branchement est appelé sortie anormale de boucle.

On a l'organigramme correspondant :



Organigramme 4.3.2

Programme PRIME 2 (version définitive):

```
0100 INPUT A,B
0105 IF INT(A/2)≠A/2 GOTO 0110
0106 A=A+1
0110 FOR N=A TO B STEP 2
0130 FOR D=3 TO SQR(N) STEP 2
0140 Q=N/D
0150 IF INT(Q)=Q GOTO 0190
0160 NEXT D
0170 PRINT N;
0190 NEXT N
0200 END
```

```
RUN
?
10,100
 11     13     17     19     23     29     31     37     41     43     47
 53     59     61     67     71     73     79     83     89     97
```

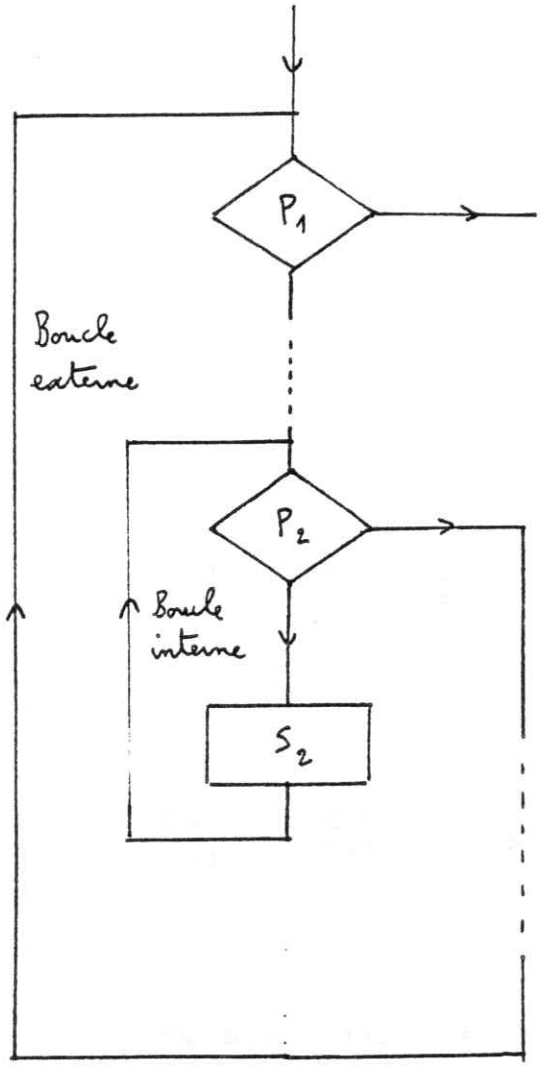
```
RUN
?
1000,1100
 1009      1013      1019      1021      1031      1033      1039
 1049      1051      1061      1063      1069      1087      1091
 1093      1097
```

Ce programme qui semble maintenant bien au point, ne fournira cependant pas exactement les résultats prévus dans quelques cas marginaux. C'est le cas pour la plupart des programmes, et il faut toujours établir soigneusement les conditions correctes d'utilisation, qui sont ici :

1) A, B entiers avec $1 \leqslant A \leqslant B$

2) L'instruction 130 réservera des surprises à l'exécution tant que $3 \leqslant \sqrt{N}$, c'est-à-dire tant que $N < 9$!. Il faudra donc analyser ce qui se passe dans les cas où $0 \leqslant A \leqslant 8$; cela dépend de la réaction du système à l'instruction 130 dans le cas où elle est contradictoire (certains systèmes exécuteront la boucle une fois, d'autres pas du tout) : cf exercice 5. Une seule chose est sûre : le seul nombre premier pair, à savoir 2, ne sera jamais imprimé par ce programme.

Ce programme est une illustration d'une structure fréquente : boucles emboitées.



On peut évidemment continuer sur ce modèle, et utiliser trois, quatre, etc... niveaux d'emboitement.

§4-4 PROGRAMMATION STRUCTUREE

Lorsqu'on écrit un programme, on souhaite évidemment qu'il fournisse les résultats désirés ; mais un bon programme doit en outre être bien structuré, pour faciliter :

- la mise au moint du programme. Lorsqu'un programme "tourne mal", c'est-à-dire ne fournit pas les résultats attendus, on doit être capable d'en découvrir rapidement l'origine. Cela n'est possible que si la structure du programme est claire. D'ailleurs un programme bien structuré contiendra d'emblée moins de fautes qu'un programme confus.

- les éventuelles modifications ultérieures. Il est fréquent qu'on ait à modifier un programme correct : pour améliorer la présentation des résultats, obtenir (ou supprimer) des résultats intermédiaires, modifier les conditions d'emploi, etc... Il est possible aussi que ces modifications soient apportées par une autre personne que l'auteur du programme. Il est donc essentiel qu'un programme soit construit de telle sorte qu'on puisse apporter des modifications, sans le faire s'écrouler comme un château de cartes.

La programmation structurée est une doctrine qui tente de donner un contenu précis à la notion de "programme bien structuré". Cette doctrine comporte de nombreux aspects, mais s'appuie sur le résultat fondamental suivant : tout organigramme peut être ramené à un enchainement de schémas répétitifs et alternatifs, tels qu'ils ont été définis dans ce chapitre.

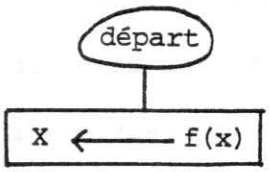
Précisons. Une procédure de calcul fournit, à partir de données x, et en utilisant des variables de calcul X, un résultat z. Pour décrire une procédure, nous avons vu

deux méthodes : organigramme, et langage type-ALGOL. Résumons-les

Organigrammes

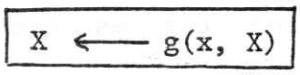
Un organigramme comprend quatre types d'instructions reliées entre elles par des flèches :

- Instruction d'entrée :

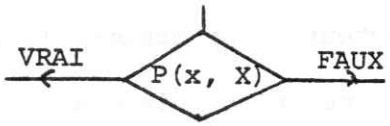


Cette instruction indique par où commencer l'exécution, et assure l'affectation de valeurs initiales aux variables de calcul (désignées par le vecteur X), en fonction du vecteur x des données.

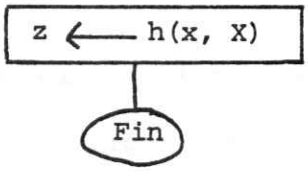
- Instructions d'affectation



- Instructions de test :
(P désigne un prédicat)



- Instructions de sortie



Un organigramme ne doit pas comporter de "cul de sac" : chaque instruction doit être située sur au moins un chemin menant de l'instruction d'entrée à l'une des instructions de sortie.

Un organigramme définit alors une fonction :

$$f : x \longrightarrow z$$

Cette fonction est définie pour les valeurs du vecteur x pour lesquelles, en exécutant l'organigramme, on atteint au bout d'un temps fini une instruction de sortie. Les autres valeurs de x sont celles pour lesquelles la procédure "boucle" -en exécutant l'organigramme, on passe sans fin par le même cycle d'instructions, sans jamais atteindre

d'instruction de sortie- : f(x) n'est alors pas défini.

Langages type-ALGOL

ALGOL est un langage qui date environ de 1960. Conçu à partir de recherches linguistiques, il a eu de nombreux successeurs qui tous permettent d'exprimer une procédure à l'aide d'un petit nombre de schémas.

Dans un langage de ce type, un programme est une suite d'instructions séparées par des points virgules : S_o ; S_1 ; S_2 ; ... ; S_n

S_o est l'instruction d'entrée :

Départ $X \longleftarrow f(x)$

qui a le même rôle que dans un organigramme. Les autres instructions peuvent être classées en cinq types :

- instructions d'affectation : $X \longleftarrow g(x, X)$
- instructions conditionnelles :

Si P(x, X) alors S sinon S'

ou bien Si P(x, X) alors S

où S et S' sont elles-mêmes des instructions (ce type de définition est récursif). P désigne un prédicat.

- instructions de répétition :

Tant que P(x, X) faire S

ou bien Répéter S jusqu'à P(x, X)

où S désigne une instruction.

- instructions de concaténation :

Début S_1 ; S_2 ; ... ; S_n fin

où S_1 , S_2 , ... , S_n désignent des instructions.

- instructions de sortie :

$z \leftarrow h(x, X)$ fin

Nous avons déjà, dans ce chapitre, présenté plusieurs programmes dans un langage de ce type. Un programme type-ALGOL définit lui aussi une fonction :

$x \longrightarrow z$

Il est évident que toute fonction définie par un programme type-ALGOL peut être définie par un organigramme. Le théorème intéressant est la réciproque :

> Toute fonction calculable par organigramme est calculable par un programme type-ALGOL.

Ce théorème n'est pas évident, puisqu'un organigramme offre apparemment plus de souplesse pour définir l'enchainement des instructions.

Nous ne démontrerons pas ce théorème. Nous indiquerons seulement plus loin quelques exemples de transformation (non immédiate) d'un organigramme en un programme type-ALGOL.

Les adeptes de la programmation structurée n'utilisent que des langages type-ALGOL. En effet, un tel langage met en évidence la structure d'un programme : alors que dans un organigramme les flèches de branchement peuvent avoir, au fond, des fonctions très différentes, dans un programme type-ALGOL les différents schémas fondamentaux-alternative répétition, concaténation- sont clairement indiqués par des instructions spéciales.

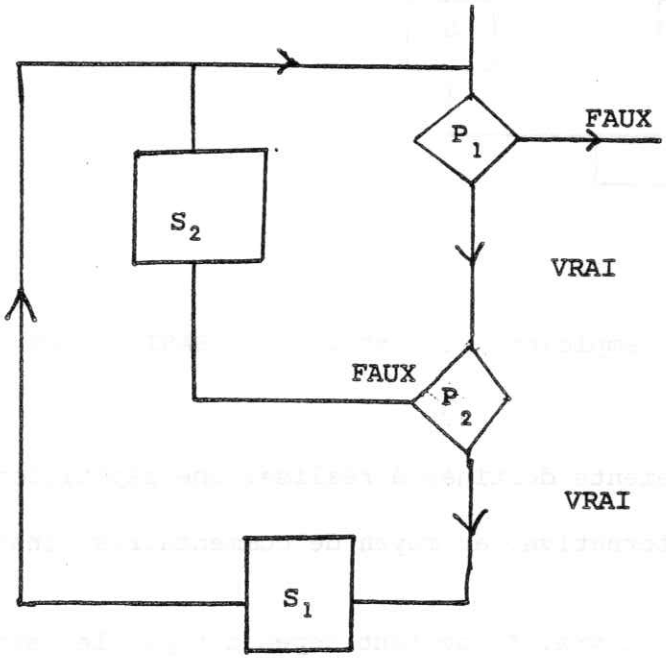
BASIC n'est pas un langage type-ALGOL. Il n'encourage donc pas de lui-même à un bon style de programmation. Il semble fructueux de s'astreindre à un certain nombre de disciplines pour que malgré tout un programme BASIC ait une structure claire.

Règle 1

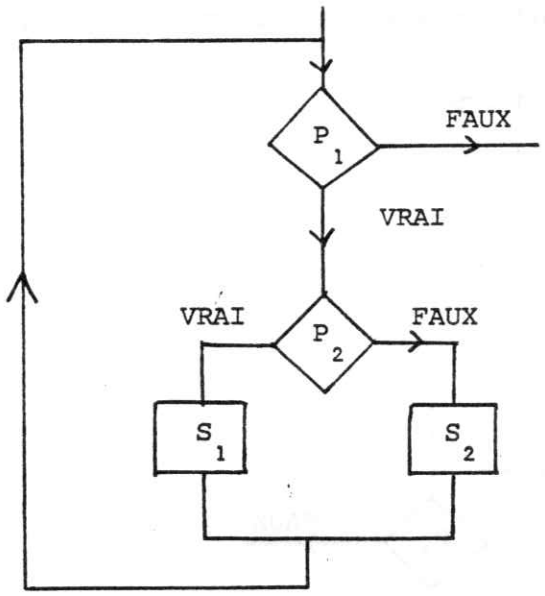
Avant d'écrire un programme BASIC, mieux vaut traduire la procédure en un langage type-ALGOL qu'en un organigramme. Ou bien alors n'utiliser que des organigrammes qui

mettent en évidence les schémas fondamentaux : alternative, répétition, concaténation.

Par exemple à un organigramme du type :



On préfèrera



Règle 2

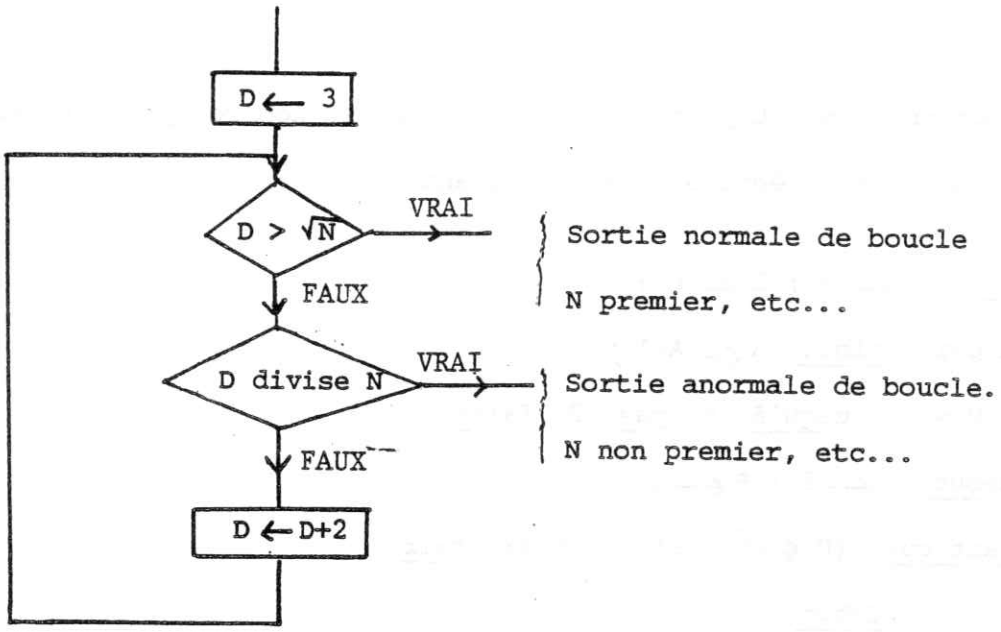
Pour exprimer une répétition, employer les instructions BASIC : FOR ... NEXT chaque fois que c'est possible.

Sinon, distinguer les branchements destinés à réaliser une répétition des branchements destinés à réaliser une alternative, au moyen de commentaires (instructions REM).

Ces règles semblent utiles à suivre. Il ne faut cependant pas les appliquer de façon dogmatique. Nous les avons respectées tout au long de ce chapitre, sauf pour la recherche de nombres premiers où nous avons employé la portion de programme PRIME :

```
0130 FOR D=3 TO SQR(N) STEP 2
0140 Q=N/D
0150 IF INT(Q)=Q GOTO 0190
0160 NEXT D
```

dont l'organigramme est donc :



Cet organigramme n'est pas la traduction d'une portion de programme type-ALGOL, car on a une boucle avec deux sorties. Voyons comment convertir cet organigramme en langage type-ALGOL ; pour cela il faut introduire une variable supplémentaire, P , dont le rôle est d'indiquer si N est premier :

P = 1 si N est premier, P = 0 sinon.

Le programme type-ALGOL correspondant peut alors s'écrire :

```
D←3 ; P←1 ;
Tant que (D ≤ √N) et (P = 1)
faire
    début
    Si D divise N alors P←0 ;
    D ← D+2
    fin ;
```

Il y a ainsi une seule sortie de boucle, et à cette sortie on sait si N est premier ou non suivant la valeur de P.

Un programme complet pour obtenir la liste des nombres premiers compris entre a et b peut alors s'écrire , en supposant a > 3 :

Départ A ← a ; B ← b ;

Si A pair alors A ← A+1 ;

Pour N = A jusqu'à B pas 2 faire

début D ← 3 ; P ← 1 ;

Tant que (D ⩽ $\sqrt{N}$) et (P = 1) faire

début

Si D divise N alors P ← 0 ;

D ← D+2

fin ;

Si P = 1 alors imprimer N †

fin ;

Fin.

La traduction BASIC sera le programme PRIME 3 :

```
0200 INPUT A,B
0210 IF INT(A/2)≠A/2 GOTO 0230
0220 A=A+1
0230 FOR N=A TO B STEP 2
0240 D=3
0250 P=1
0260 N1=SQR(N)
0270 REM BOUCLE TANT QUE (D≤N1) ET (P=1)
0280 IF D>N1|P=0 GOTO 0340
0290 Q=N/D
0300 IF INT(Q)≠Q GOTO 0320
0310 P=0
0320 D=D+2
0330 GOTO 0270
0340 IF P=0 GOTO 0360
0350 PRINT N;
0360 NEXT N
0370 END
```

N.B. Le signe | signifie 'ou' sur IBM 5100.

† Si l'on désire suivre la règle qui consiste à désigner par z le vecteur résultat de la procédure, il faut remplacer 'imprimer N' par :

z ← z, N (et rajouter : z ← vide au départ).

Il n'est vraiment pas certain que ce programme soit préférable au programme PRIME 2 du paragraphe 4.3.2 : on s'est privé d'une boucle FOR ... NEXT, et on a dû introduire une variable supplémentaire P, dont le rôle n'est pas évident pour celui qui lit le programme (on peut évidemment expliquer le rôle de P par des lignes de commentaires). Par contre on conçoit que les boucles à sorties multiples gênent l'analyse théorique (par exemple la preuve) d'un programme.

Remarque

Ce serait une erreur de supprimer l'instruction 260 et de remplacer 280 par :

```
280 IF D>SQR(N) | P=0 GOTO 340
```

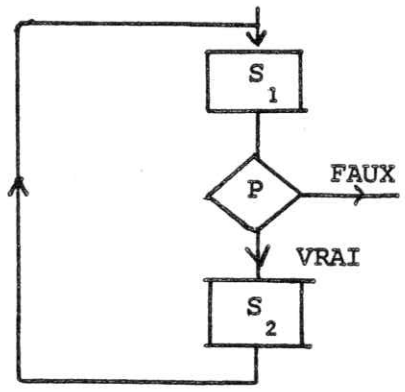
car alors l'ordinateur est obligé de calculer SQR(N) à chaque exécution de la boucle interne 270-330.

Cet inconvénient n'existait pas avec le programme PRIME, car lors de l'exécution de la boucle :

```
130 FOR D=3 TO SQR(N) STEP 2
      :
160 NEXT D
```

la valeur de SQR(N) est calculée une fois et une seule par l'ordinateur.

Terminons sur un autre cas, assez fréquent, où suivre les règles de la programmation structurée n'est pas tout à fait évident. Il s'agit des boucles avec "test au milieu" du type :



La traduction en langage type-ALGOL sera :

S_1 ; <u>tant que</u> P <u>faire</u> <u>début</u>

S_2 ; S_1

<u>fin</u>

Ceci oblige à écrire deux fois le bloc d'instructions S_1 ; l'avantage est de montrer clairement que S_1 est <u>toujours</u> exécuté une fois de plus que S_2 ; et de montrer l'ordre réel d'exécution des blocs S_1, S_2 à l'intérieur de la boucle.

Exercices

1) Dans l'exemple 4.2.3, supposons que par mégarde, on intervertit les premières lignes :

```
0100 INPUT N
0110 FOR J=1 TO N
0120 S1=0
0130 S2=0
0140 S1=S1+1/J
0150 S2=S2+1/(J*J)
0160 NEXT J
0170 PRINT S1,S2
0180 END
```

Quel seront les résultats imprimés par l'ordinateur lors de l'exécution ?

2) Si dans l'exemple 4.2.3, on intervertit les lignes 160 et 170, quel est le résultat de l'exécution du programme ?

3) Lors de l'exécution du programme FACT, on entre une valeur positive non entière, en réponse à l'instruction :

```
100 INPUT N
```

Quel est le résultat du calcul ?

4) Construire un programme qui vous permette de savoir quelle valeur est attribuée à l'indice d'une boucle par votre ordinateur après exécution complète d'une boucle du type :

```
FOR K=1 TO 10
```

(sur certains systèmes, on a K=10 <u>après</u> exécution de la boucle, sur d'autres on a : K=11)

Construire aussi un programme qui permette de savoir si votre ordinateur exécute une fois ou pas du tout une boucle illogique (du type : FOR K=10 TO 1).

5) Analyser les résultats du programme PRIME 2 dans le cas a = 1 ; on fera successivement les hypothèses suivantes :

- le système exécute une fois et une seule une boucle "illogique"

- le système n'exécute pas les boucles illogiques

6) Le programme FACT fournit comme résultat imprimé la valeur de N !

Modifier ce programme afin d'obtenir comme résultats imprimés les valeurs successives de 2 !, 3 !, 4 !, ... N !

7) Modifier le programme RACINE afin que les sorties imprimées soient les suivantes, si l'utilisateur entre 5 comme valeur de A et 10^{-2} comme valeur de E :

```
VALEUR APPROCHEE DE RACINE CARREE DE 5     A 1E-2     PRES: 2.238095
VALEUR DE LA FONCTION BASIC SQR: 2.236068
```

8) Equations du 2è degré

$$ax^2 + bx + c = 0$$

(coefficients réels).

Ecrire un programme qui donne des résultats imprimés du type suivant :

```
RUN
COEFFICIENTS
?
1,-3,2
RACINES: 2                    1

RUN
COEFFICIENTS
?
1,1,1
PAS DE RACINE REELLE



RUN
COEFFICIENTS
?
1,2,1
RACINE DOUBLE:-1
```

9) Ecrire un programme qui calcule et imprime la valeur de :

$$C_n^k = \binom{n}{k} = \frac{n(n-1)(n-2) \ldots (n-k+1)}{k(k-1)(k-2)\ldots \times 1}$$

(n et k seront entrés par l'utilisateur en réponse à une instruction INPUT) ; modifier ce programme afin d'obtenir, avec le moins possible de calculs, l'impression de la $n^{ème}$ ligne du triangle de Pascal (n sera entré en réponse à une instruction INPUT) :

$$C_n^0 , C_n^1 , C_n^2 \ldots C_n^n$$

Modifier à nouveau ce programme pour obtenir cette fois les n premières lignes du triangle de Pascal.

10) Prouver la correction de l'organigramme 4.3.1

Activités

1) Dérivation

Construire un programme qui donne les valeurs du rapport :

$$\frac{f(x+h) - f(x)}{h}$$ pour différentes valeurs de h de plus en plus petites (on peut par exemple diviser chaque fois h par 2, ou par 10). On prendra pour f la fonction de son choix ; la valeur de x pourra être entrée en réponse à une instruction INPUT. Comment expliquez-vous les résultats (qui sont probablement plus compliqués qu'on ne le penserait à première vue) ?

Recommencez en calculant :

$$\frac{f(x+h) - f(x-h)}{2h}$$

2) Séries

Le calcul de sommes de séries est un champ d'activités assez vaste.

On sait par exemple que $\sum_{1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$. On peut utiliser l'ordinateur pour vérifier que les sommes partielles $\sum_{1}^{n} \frac{1}{k^2}$ se rapprochent effectivement de la valeur $\frac{\pi^2}{6}$, et évaluer les restes $\sum_{n}^{\infty} \frac{1}{k^2}$ (attention au problème de l'accumulation des erreurs d'arrondi ; cf remarque suivant l'exemple 4.2.3, section 4.2).

On peut aussi comparer $\sum_{o}^{n} \frac{x^k}{k!}$ avec e^x, fourni par la fonction BASIC EXP.

Idem pour sin x, cos x, etc ... Pour les développements en série de certaines fonctions, on rencontrera évidemment des problèmes de rapidité de convergence (cf chapitre 8).

5 VECTEURS ET MATRICES

Les procédures qui n'utilisent que des variables scalaires s'avèrent rapidement insuffisantes pour résoudre de larges catégories de problèmes : résolution d'un système de n équations à n inconnues, traitement de statistiques, etc Il est alors nécessaire d'utiliser comme variables des vecteurs ou des matrices de façon générale on parle alors de variables indicées (par opposition aux variables scalaires), ou de tableaux.

§ 5-1 DECLARATION DES TABLEAUX

Les noms autorisés en BASIC pour les tableaux sont simplement :

A,B,C,.....,Z

Si V, par exemple, désigne un vecteur, c'est-à-dire une suite de scalaires, chacun d'eux est repéré par son indice : le troisième élément du vecteur V, par exemple, est désigné par :

V(3)

Il est clair que la notation indicielle habituelle V_3 n'est pas adaptée à l'utilisation d'un clavier pour communiquer avec l'ordinateur. Les valeurs autorisées pour l'indice sont définies, en BASIC comme dans la plupart des langages de programmation, par une instruction de déclaration, qui doit précéder, dans le programme, tout emploi de la variable V ; une telle instruction est repérée par le mot DIM.

Par exemple :

```
100  DIM  V(50)
```

est une instruction de déclaration qui signifie que V désigne dans la suite du programme un vecteur, et que les indices pourront varier entre 1 et 50. Ce vecteur comportera donc au plus 50 éléments.

La forme générale de l'instruction est :

```
DIM  V(n )
```

où n est une constante. Un programme qui commence par :

```
100   INPUT   N
110   DIM   V(N)
```

n'est donc pas valide. (Sur la plupart des systèmes).

Pour les matrices, chaque élément est repéré par deux indices. Par exempl

```
100   DIM   M(5,20)
```

est une instruction de déclaration qui signifie que M désigne dans la suite programme une matrice à 5 lignes et 20 colonnes. L'élément de la 2ème ligne e de la 6ème colonne sera désigné par :

```
M(2,6)
```

(à la place de la notation traditionnelle, mais impraticable sur clavier, $M_{2,}$

Une même instruction de déclaration peut servir à déclarer plusieurs vecteurs ou matrices.

Exemple :

```
100   DIM   A(20)   ,   M(5,20)   ,   V(50)
```

déclare deux vecteurs et une matrice.

Par contre une variable indicée ne peut être déclarée qu'une fois dans un programme : on ne peut pas modifier la déclaration d'une variable déjà déclar comme vecteur ou matrice.

ATTENTION : En BASIC standard, l'indice 0 n'est pas admis pour désigner un él ment d'un vecteur ou d'une matrice. En général, un indice doit au être inférieur à 255 (pour pouvoir être stocké par l'ordinateur s un octet).

Remarque : Il existe en BASIC une facilité curieuse, qui en fait n'en est pas une : on peut "oublier" de déclarer un vecteur ou une matrice.

L'interpréteur supplée à cet oubli en déclarant par défaut le vecte de dimension 10, et la matrice de dimension 10 x 10.

L'inconvénient de ce genre de "facilité" est qu'une instruction comme :

```
X  =  A (B + C)
```

est interprétée comme faisant référence à un vecteur A déclaré par défaut, alo que bien souvent on voulait seulement écrire :

```
X = A * (B + C)     !
```

Une fois un vecteur (ou une matrice) déclaré (e), ses éléments peuvent être utilisés partout où une variable scalaire a le droit de l'être. Les indices peuvent être n'importe quelle expression numérique valide.

§ 5-2 EXEMPLE : REPARTITION D'UNE SUITE

Pour illustrer l'emploi de variables indicées, écrivons un programme qui permette d'étudier la répartition des nombres aléatoires fournis par la fonction RND. Ces nombres sont, on le sait, compris entre 0 et 1. Découpons le segment $[0 \quad 1]$ en p segments égaux :

et cherchons, sur n valeurs fournies par la fonction RND, combien appartiennent à S_1, combien à S_2, ..., combien à S_p. Le résultat du programme sera donc un vecteur Z , Z(i) désignant le nombre de valeurs appartenant à S_i. Pour écrire le programme, il est utile de remarque que si $X \in [0 \quad 1]$, l'indice i de sa classe S_i est donné par la formule BASIC :

```
I = 1 + INT (P * X)
```

D'où le programme REPARTITION :

```
0100 DIM Z(50)
0110 PRINT 'NOMBRE DE CLASSES';
0120 INPUT P
0130 FOR I=1 TO P
0140 Z(I)=0
0150 NEXT I
0160 PRINT 'NOMBRE D''EXPERIENCES';
0170 INPUT N
0180 FOR K=1 TO N
0190 I=1+INT(P*RND)
0200 Z(I)=Z(I)+1
0210 NEXT K
0220 PRINT 'STATISTIQUE:'
0230 FOR I=1 TO P
0240 PRINT Z(I);
0250 NEXT I
0260 PRINT
0270 END
RUN
NOMBRE DE CLASSES
?
10
NOMBRE D'EXPERIENCES
?
500
STATISTIQUE:
```

Vu l'instruction 100 DIM Z(50), ce programme ne fonctionne que pour u nombre de classes inférieur ou égal à 50. Au-delà, il n'y aura apparition d'u message d'erreur à l'éxécution de l'instruction 140, lorsque I dépassera 50.

Les instructions 130 à 150 servent d'initialisation pour le vecteur Z. En fait, certains systèmes assurent automatiquement l'initialisation à zéro, mai c'est une mauvaise habitude de compter là-dessus.

Les instructions 230 à 260 sont elles aussi standard, pour assurer l'impr sion (en ligne) d'un vecteur. L'instruction 260 assure le retour à la ligne d la tête d'impression, après l'impression de Z(p) ; comme ici le programme se termine après cette impression, on pourrait se passer de la ligne 260, mais c'est une bonne habitude que de ne pas l'oublier.

On remarquera la forte affinité des vecteurs pour les boucles FOR .... N En fait, la structure de vecteur pour les données correspond à la structure d boucle FOR .... NEXT pour les programmes.

§ 5-3 INSTRUCTIONS MAT

BASIC fournit des instructions spéciales pour certaines boucles standard FOR .... NEXT utilisées dans le traitement des vecteurs (ou des matrices). Ces instructions sont caractérisées par le mot MAT, et varient légèrement sui vant les systèmes.

Par exemple, sur IBM 5100, on peut remplacer les lignes 130 à 150 du prog me REPARTITION par une seule instruction :

```
130 MAT  Z  =  (0)
```

à la nuance près que cette instruction initialise à zéro tous les éléments d Z †, pas seulement les P premiers.

De même les lignes 230 à 260 peuvent être remplacées par :

```
230  MAT  PRINT  Z  ;
```

---

† C'est-à-dire ici les 50 éléments de Z, vu l'instruction DIM Z(50)

Mais là aussi, ce sont les 50 éléments du vecteur Z qui seront imprimés. Il existe, pour éviter cet inconvénient, des moyens de redimensionner des vecteurs (ou des matrices) à l'intérieur d'instructions MAT d'affectation.

Nous ne développerons pas l'étude de ces instructions, qui sont de simples facilités et n'apportent rien de fondamentalement nouveau †. Consultez le manuel de référence de votre système si vous vous servez suffisamment souvent de BASIC pour que cela vaille la peine d'assimiler la syntaxe et le rôle de ces instructions.

---

† Certaines de ces instructions permettent (cela dépend des systèmes) de faire directement des sommes ou des produits de matrices.

§ 5-4 EXEMPLE DE SIMULATION - HISTOGRAMME

Continuons par un exemple de simulation : lancer simultané de deux dés, avec comptage de la somme des points obtenus, et représentation en histogramm de la statistique des résultats.

Pour simuler le lancer d'un dé, on utilise la fonction RND†, qui fournit un nombre au hasard compris (strictement) entre 0 et 1. Donc

```
6 * RND
```

est compris (strictement) entre 0 et 6, et

```
INT (6 * RND)
```

est un entier au hasard compris entre 0 et 5, et l'instruction:

```
I = 1 + INT (6 * RND)
```

attribue à I la valeur qu'on obtiendrait en lançant un dé au hasard!

---

† Rappel: la syntaxe de la fonction RND varie suivant les systèmes.

Programme JEU DE DES

```
0100 DIM Z(12)
0110 MAT Z=(0)
0120 PRINT 'NOMBRE D''EXPERIENCES';
0130 INPUT N
0140 REM -------SIMULATION--------
0150 FOR K=1 TO N
0160 I=1+INT(6*RND)
0170 I=I+1+INT(6*RND)
0180 Z(I)=Z(I)+1
0190 NEXT K
0195 PRINT 'STATISTIQUE:'
0200 FOR I=2 TO 12
0210 PRINT Z(I);
0220 NEXT I
0230 PRINT
0240 REM -------HISTOGRAMME-------
0250 FOR I=2 TO 12
0260 FOR K=1 TO Z(I)
0270 PRINT '*';
0280 NEXT K
0290 PRINT
0300 NEXT I
0310 END

RUN
NOMBRE D'EXPERIENCES
?
 300
STATISTIQUE:
 15     14     19     27     31     50     56     35     29     15     9
**************
**************
*******************
***************************
*****************************
*************************************************************
*************************************************************************
*************************************
*****************************
***************
**********
```

Pour l'histogramme, ne pas oublier l'instruction 290 ! Ni d'ailleurs l'instruction 230.

Voici ce qu'on obtient en oubliant l'instruction 230

```
RUN
NOMBRE D'EXPERIENCES
?
 300
STATISTIQUE:
 15    14    19    27    31    50    56    35    29    15    9    **********
***************
********************
****************************
******************************
**************************************************
**************************************************
**********************************
****************************
***************
*********
```

Et en oubliant aussi l'instruction 290 :

```
RUN
NOMBRE D'EXPERIENCES
?
100
STATISTIQUE:
    5     7     5     9     7    17    23     7    10     6     4**********
**************************************************************************
****************************
```

Décevant !

+ + + + + + + + +

Nous terminerons par deux exemples d'utilisation de matrices :

- calcul de moyennes, scolaires par exemple

- Triangle de Pascal

5-5 CALCUL DE MOYENNES

Partons d'un tableau de notes obtenues par huit élèves à cinq devoirs

Devoirs / Elèves	1	2	3	4	5
1	12,5	8	13	14,5	6
2	6	8	5	9,5	4
3	13	12	14,5	12	10
4	16	12	13,5	14	12
5	7	4	8	2,5	5
6	6	3	5,5	7,5	4
7	12	10,5	13	14	8,5
8	4	6	10	8	9

Ecrivons un programme qui nous fournisse lesmoyennes de chaque élève, puis les moyennes par devoir. Sur le tableau, cela correspond à des moyennes par lignes, puis par colonnes. De façon générale, nous supposerons que le tableau porte sur n élèves et p devoirs. Les valeurs de n et p, ainsi que le tableau des notes, sont fournies par des instructions DATA.

Programme MOYENNES

```
0100 DATA 8,5
0110 DATA 12.5,8,13,14.5,6
0120 DATA 6,8,5,9.5,4
0130 DATA 13,12,14.5,12,10
0140 DATA 16,12,13.5,14,12
0150 DATA 7,4,8,2.5,5
0160 DATA 6,3,5.5,7.5,4
0170 DATA 12,10.5,13,14,8.5
0180 DATA 4,6,10,8,9
0500 DIM T(30,10),E(30),D(10)
0510 READ N,P
0520 REM ------LECTURE DES NOTES------
0530 FOR I=1 TO N
0540 FOR J=1 TO P
0550 READ T(I,J)
0560 NEXT J
0570 NEXT I
0580 REM ------MOYENNES PAR ELEVES(LIGNES)------
0590 FOR I=1 TO N
0600 E(I)=0
0610 FOR J=1 TO P
0620 E(I)=E(I)+T(I,J)
0630 NEXT J
0640 E(I)=E(I)/P
0650 NEXT I
0660 REM ------MOYENNES PAR DEVOIRS(COLONNES)------
0670 FOR J=1 TO P
0680 D(J)=0
0690 FOR I=1 TO N
0700 D(J)=D(J)+T(I,J)
0710 NEXT I
0720 D(J)=D(J)/N
0730 NEXT J
0740 REM ------IMPRESSION DES RESULTATS-----
0750 PRINT 'MOYENNES PAR ELEVES:'
0760 FOR I=1 TO N
0770 PRINT USING 0850,E(I);
0780 NEXT I
0790 PRINT
0800 PRINT 'MOYENNES PAR DEVOIRS:'
0810 FOR J=1 TO P
0820 PRINT USING 0850,D(J);
0830 NEXT J
0840 PRINT
0850 :  ##.#
0860 END
```

```
RUN
MOYENNES PAR ELEVES:
  10.8    6.5   12.3   13.5    5.3    5.2   11.6    7.4
MOYENNES PAR DEVOIRS:
   9.6    7.9   10.3   10.3    7.3
```

On voit que de même que les vecteurs affectionnent les boucles, les matrice affectionnent les doubles boucles.

Il est utile de bien saisir le mécanisme de traitement d'une matrice par lignes (boucle 530-570 : lecture des notes, et boucle 590-650 : moyennes par élèves) et celui du traitement par colonnes (boucle 670-730) ; le deuxième cas se déduit du premier en échangeant boucle interne et boucle externe.

L'impression des résultats a été séparée, dans le programme MOYENNES, de leur calcul ; on utilise ainsi quelques lignes de programme supplémentaires, mais la méthode est beaucoup plus sûre, et le programme mieux structuré.

Remarques : 1) Vu l'instruction 500, le programme fonctionne tant que le nombre d'élèves est inférieur à 30, et nombre de devoirs à 10.
Pour modifier les données, il faut bien sûr modifier les lignes DATA. On a fait commencer la numérotation du programme proprement dit à 500 pour permettre sans risque d'erreur l'insertion de nombreuses lignes DATA en début de programme.

2) L'utilisation de PRINTUSING permet d'éviter les décimales non significatives

5-6 TRIANGLE DE PASCAL

On sait que, si l'on désigne par C_n^p le nombre de parties à p éléments incluses dans un ensemble de n éléments, on a la formule remarquable :

$$(1) \qquad C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$$

Cette formule permet de construire le proche en proche le fameux "triangle de Pascal" :

n \ p	0	1	2	3	4	5	.....
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	

Avant d'écrire un programme à partir de la formule (1) il faut en préciser soigneusement les conditions de validité, qui sont :

$$n \geqslant 2$$
$$1 \leqslant p \leqslant n-1$$

En particulier, (1) n'est pas valable pour $p = 0$ et $p = n$ †

La construction du triangle de Pascal par un programme BASIC présente d'autre part un petit piège, car l'indice 0 est interdit en BASIC -standard. Le coefficient du binôme C_n^p doit donc être représenté par l'élément C(N+1 , P+1) de matrice C. D'où le programme :

Programme PASCAL

```
0100 DIM C(21,21)
0110 INPUT T
0120 FOR N=1 TO T+1
0130 C(N,1)=1
0140 C(N,N)=1
0150 NEXT N
0160 FOR N=3 TO T+1
0170 FOR P=2 TO N
0180 C(N,P)=C(N-1,P)+C(N-1,P-1)
0190 NEXT P
0200 NEXT N
0210 FOR N=1 TO T+1
0220 FOR P=1 TO N
0230 PRINT C(N,P);
0240 NEXT P
0250 PRINT
0260 NEXT N
0270 END

RUN
?
8
 1
 1      1
 1      2      1
 1      3      3      1
 1      4      6      4      1
 1      5      10     10     5      1
 1      6      15     20     15     6      1
 1      7      21     35     35     21     7      1
 1      8      28     56     70     56     28     8      1
```

† On pourrait aussi convenir : $C_n^{-1} = C_n^{n+1} = 0$
pour élargir le champ de validité de la formule (1)

Ce programme fonctionne pour les valeurs de T inférieures ou égales à 20. Insistons à nouveau sur l'utilité, pour éviter les erreurs, de séparer le calcul du tableau et son impression, même si cela coûte quelques lignes supplémentaires.

Si l'on veut un beau triangle, ne pas oublier l'instruction 250 PRINT !

Il faut aussi être très soigneux dans le choix des bornes de variations pour les boucles FOR... NEXT (instructions 120, 160, 170, 210, 220) : l'instruction

```
FOR N=3 TO T+1
```

n'a pas le même effet que l'instruction :

```
FOR N=1 TO T          !
```

C'est évident, mais on a souvent tendance à penser que l'ordinateur fera de lui-même les petits ajustements nécessaires ; les ordinateurs ne sont hélas pas encore doués de bon sens, ou en tout cas pas du même que nous.

Exercices du chapitre 5

1) Ecrire un programme qui construise le vecteur P des nombres premiers compris entre 2 et n ; pour déterminer si un entier est premier ou non, on ne testera que les diviseurs premiers, en utilisant les éléments déjà disponibles de P.

Comparer la rapidité d'exécution avec celle du programme PRIME 2 (section 4.3.2)

2) Compléter le programme précédent par un programme de décomposition en facteurs premiers, valable pour les entiers inférieurs à n^2, en utilisant le vecteur P. Comparer la rapidité d'exécution avec celle du programme DFP (section 4.3.1).

6 SOUS-PROGRAMMES

La notion de sous-programme constitue un concept clef en informatique, car la meilleure méthode pour mener à bien une tâche complexe reste de la décomposer en tâches plus simples ! La réalisation d'un programme complexe passe donc par l'utilisation de sous-programmes appropriés. En principe, chacune de ces procédures annexes peut être mise au point et testée séparément, avant d'être utilisée à l'intérieur du programme principal. Et cette organisation hiérarchique n'est pas nécessaire: plusieurs procédures peuvent collaborer 'démocratiquement' à la réa-lisation d'une tâche complexe, sans qu'il y ait vraiment de procédure principale. La collaboration entre ces procédures est assurée par la possibilité,pour chaque procédure, d'utiliser les résultats fournis par d'autres.

Chacune de ces procédures peut être programmée:

- soit successivement par une même personne; l'avantage est alors de s'atta-quer aux difficultés une à une.
- soit par différentes personnes, ce qui permet le travail en équipe; en outre les procédures d'usage courant peuvent être stockées et former une bibliothèque à la disposition des utilisateurs.

Hélas BASIC n'est guère conçu pour permettre à un programme d'en appeler un autre, c'est à dire d'en déclencher l'exécution pour en utiliser les résultats. Les sous-programmes BASIC font partie du programme principal, et permettent seu-lement de rendre plus claire la structure du programme principal - ou d'éviter des répétitions. A ce titre leur emploi est à recommander; mais l'absence en BASIC des mécanismes† qui permettent la collaboration entre procédures indépendantes est peut-être la plus grave faiblesse de ce langage.

Il existe deux sortes d'instructions BASIC pour l'utilisation de sous-programmes:

- En général on utilise les instructions GOSUB et RETURN (section 6.1)
- Certaines fonctions peuvent être définies au moyen de l'instruction DEF (section 6.2).

Nous n'étudions ici que le BASIC standard; certaines versions de BASIC possèdent des formes un peu plus raffinées de sous-programmes: consultez le manuel de référence de votre système.

---

†Du coup nous n'étudierons pratiquement pas ces mécanismes, malgré leur impor-tance en informatique. Nous utiliserons cependant des procédures récursives (c'est à dire qui s'utilisent ... elles-mêmes), en particulier au chapitre 12.

§6.1 Sous-programmes appelés par GOSUB

Un sous - programme BASIC est une suite d'instructions terminée par l'instruc--tion: RETURN. L'exécution de ce sous - programme peut être déclenchée à l'intérieur du programme principal par l'instruction:

GOSUB n

où n est le numéro de la première instruction du sous - programme.

Exemple.

Le programme PRIME 2 (section 4.3.2) de recherche des nombres premiers compris entre a et b peut être réécrit plus clairement en utilisant un sous - programme. En effet rappelons sa structure:

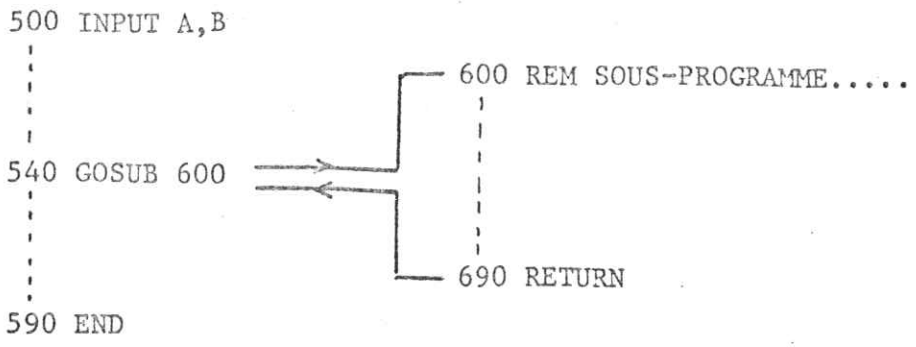
Départ A ← a; B ← b;
Si A pair alors A ← A + 1;
Pour N=A jusqu'à B pas 2 faire
    si 'N premier' alors imprimer N;
Fin.

En introduisant un sous - programme qui fournit comme résultat P=1 si N est premier et P=0 sinon, on obtient le programme BASIC suivant:

Programme PRIME 4

```
0500 INPUT A,B
0510 IF INT(A/2)≠A/2 GOTO 0530
0520 A=A+1
0530 FOR N=A TO B STEP 2
0540 GOSUB 0600
0550 IF P=0 GOTO 0570
0560 PRINT N;
0570 NEXT N
0580 PRINT
0590 END
0600 REM SOUS-PROGRAMME  P← SI N PREMIER ALORS 1 SINON 0
0610 REM HYPOTHESE: N IMPAIR
0620 FOR D=3 TO SQR(N) STEP 2
0630 Q=N/D
0640 IF INT(Q)=Q GOTO 0680
0650 NEXT D
0660 P=1
0670 GOTO 0690
0680 P=0
0690 RETURN
```

L'exécution peut être représentée par le schéma suivant:



Le sens précis de l'instruction RETURN est donc: 'Retourner à l'instruction qui suit l'instruction GOSUB (ici: l'instruction 550 IF P=0....)'.

PRIME 4 est plus long que PRIME 2; par contre sa structure est plus claire, et il devient évident que PRIME 4 est correct pourvu que le sous - programme qui débute à la ligne 600 le soit. D'autre part on peut, sans toucher au programme principal, modifier le sous - programme qui débute à la ligne 600†, soit pour corriger une erreur éventuelle, soit pour l'améliorer.

Attention: Ne pas oublier, dans le programme principal, l'instruction:

590 END

sinon l'exécution continue par une nouvelle exécution (imprévue) du sous - programme, avec un message d'erreur quand on arrive à l'instruction RETURN (car le sous - programme n'a été appelé par aucun GOSUB).

---

†L'un des inconvénients des sous - programmes BASIC est de ne pas avoir de nom!

L'exemple précédent vise principalement à illustrer l'emploi de sous - programm comme le programme PRIME 2 est simple, l'emploi d'un sous - programme dans ce cas reste une affaire de goût. Le volume 2 de cette brochure contiendra des exemples de programmes plus complexes, où l'emploi de sous - programmes est presque indispen--sable. L'appel de sous - programmes par GOSUB peut être utilisé de façon répétée; en particulier:

- un sous - programme donné peut être appelé plusieurs fois par le programm principal; c'est d'ailleurs un cas où l'utilisation de sous - programmes s'impose, puisqu'elle permet d'éviter des répétitions.
- un sous - programme peut appeler ses propres sous - programmes, etc.... Le niveau d'emboitement des appels n'est limité que par les caractéris--tiques de l'interpréteur: consultez le manuel de référence de votre système.

Lorsqu'il rencontre une instruction RETURN, l'interpréteur effectue un branche--ment à l'instruction qui suit l'instruction GOSUB qui a appelé le sous - programme On peut se demander comment il s'y retrouve dans le cas d'appels répétés, ou de sous - programmes emboités. La solution passe par l'utilisation d'une pile de stockage des appels. Chaque fois que l'interpréteur exécute une instruction GOSUB, il stocke le numéro de cette instruction GOSUB dans la pile, à la suite (ou, pour reprendre l'analogie suggérée par le terme de pile, 'au-dessus') des numéros déjà stockés; inversement, chaque fois que l'interpréteur exécute une instruction RETURN il va lire le dernier numéro stocké (celui du 'dessus' de la pile), pour savoir où effectuer le branchement de retour, puis il efface ce numéro de la pile; en d'autres termes, il retire de la pile l'élément supérieur, de la même façon qu'on retire une assiette d'une pile d'assiettes; une pile de stockage fonctionne sur le principe 'dernier arrivé, premier sorti'. Une simulation manuelle de ce syst ème sur quelques exemples montre qu'il résoud correctement le problème du couplage des instructions GOSUB et RETURN.

Le couple d'instructions GOSUB et RETURN permet d'effectuer des branchements, comme une instruction GOTO, mais avec l'avantage décisif que l'utilisateur n'a pas à se soucier des 'adresses de retour': c'est l'interpréteur qui s'en charge, grâce au système décrit ci-dessus.

§6.2 Instruction DEF

Cette instruction permet à l'utilisateur de définir ses propres fonctions, puis de les employer comme des fonctions BASIC incorporées (cf. section 2.3). On peut ainsi définir jusqu'à 26 fonctions, appelées:

FNA, FNB, FNC, ... , FNZ

Par exemple l'instruction:

```
0100 DEF FNA(X)=3*X↑2-4*X+1
```

définit la fonction FNA: $x \longmapsto 3x^2-4x+1$, et cette fonction peut ensuite être employée au même titre et avec la même syntaxe que les fonctions SIN, INT, etc... En BASIC standard, la définition de FNA doit tenir sur une ligne.

La variable X qui intervient dans l'instruction DEF est, par exception, muette, et n'a rien à voir avec un éventuel registre de mémoire appelé par ailleurs dans le programme X. D'ailleurs, la définition:

```
0100 DEF FNA(X)=3*X↑2-4*X+1
```

est équivalente à:

```
0100 DEF FNA(T)=3*T↑2-4*T+1
```

X (ou T dans le deuxième cas) est appelé paramètre dans la définition de fonction. Si plus loin dans le programme on évalue par exemple FNA(2*B+1), le mécanisme d'éva--luation est le suivant: la valeur actuelle de 2*B+1 est 'passée'[†] au paramètre qui figure dans la définition de FNA, puis l'évaluation a lieu conformément à la définition de FNA. Si le programme utilise par ailleurs une variable de même nom que le paramètre, la valeur de cette variable n'est pas modifiée au cours du pro--cessus d'évaluation de FNA.

Exemple: Recherche d'une racine d'une équation f(x)=0 par dichotomie.

On sait que si f est continue sur $[a\ b]$ et si f(a) et f(b) sont de signes contraires, alors f admet au moins une racine sur $[a\ b]$. D'où une procédure de recherche d'une[††] de ces racines, par dichotomie:

[†] Le passage de valeurs aux paramètres d'une procédure est le moyen essentiel de communication entre procédures d'un 'bon' langage. En BASIC standard, il n'existe que pour les fonctions incorporées, ou définies par DEF.

[††] En pratique on s'assure que f est aussi monotone sur $[a\ b]$, de telle sorte que la racine est unique.

Départ A ← a; B ← b;
Tant que |B-A|>ε faire
  début C ← $\frac{A+B}{2}$;
  si f(A).f(C) < 0 alors B ← C
                   sinon A ← C
  fin;
z ← A
Fin

Le programme BASIC correspondant doit spécifier la fonction f; si on prend par exemple, comme précédemment, f: $x \longmapsto 3x^2-4x+1$ on obtient:

Programme DICHO

```
0100 DEF FNA(X)=3*X↑2-4*X+1
0110 PRINT 'BORNES A,B DE L''INTERVALLE';
0120 INPUT A,B
0130 PRINT 'PRECISION';
0140 INPUT E
0150 REM BOUCLE TANT QUE !B-A!>E
0160 IF ABS(B-A)≤E GOTO 0230
0170 C=(A+B)/2
0180 IF FNA(A)*FNA(C)>0 GOTO 0210
0190 B=C
0200 GOTO 0150
0210 A=C
0220 GOTO 0150
0230 PRINT 'RACINE A ';E;' PRES: ';A
0240 END

RUN
BORNES A,B DE L'INTERVALLE
?
 .5    2
PRECISION
?
 1E-5
RACINE A  1E-5      PRES:   .999998


RUN RD=10
BORNES A,B DE L'INTERVALLE
?
0,0.5
PRECISION
?
1E-8
RACINE A  1E-8      PRES:   .3333333284


 READY
```

Sur IBM 5100, cette commande précise que l'o désire une impression de résultats avec 10 chiffr significatifs.

Ce programme ne fonctionne correctement que si f(a) et f(b) sont effectivement de signes contraires. Pour prévoir le cas de données incorrectes, il suffit de rajou-ter par exemple les instructions:

```
0125 IF FNA(A)*FNA(B)>0 GOTO 0300
0300 PRINT 'ERREUR; F(A) ET F(B) DE MEME SIGNE:'
0310 PRINT FNA(A),FNA(B)
0320 END
```

Exemple d'exécution:

```
RUN
BORNES A,B DE L'INTERVALLE
?
0,2
ERREUR; F(A) ET F(B) DE MEME SIGNE:
 1                  5

 READY
```

Si l'on désire changer la fonction f, il suffit de changer la ligne 100 du programme. Exemple:

```
100 DEF FNA(X)=LOG(X)-1

RUN RD=8
BORNES A,B DE L'INTERVALLE
?
1,10
PRECISION
?
1E-8
RACINE A  1E-8      PRES:  2.71828183

PRINT EXP(1)
 2.71828183
```

Ça marche!

Remarque: Si le système possède la fonction incorporée SGN (signe), définie par:

SGN: $x \longmapsto -1$ si $x < 0$

$x \longmapsto 0$ si $x = 0$

$x \longmapsto 1$ si $x > 0$,

il vaut mieux remplacer l'instruction 180 par:

```
0180 IF SGN(FNA(A))=SGN(FNA(C)) GOTO 0210
```

pour éviter le calcul exact d'un produit bien inutile.... Idem pour l'instruction 125.

Sur IBM 5100, on peut définir des fonctions à plusieurs arguments, et la définiti d'une fonction peut s'étendre sur plusieurs lignes. Le programme PRIME 4[†] peut par exemple être réécrit en utilisant une fonction FNP définie par:

FNP(u) = si u premier alors 1 sinon 0 (u est supposé impair)

Programme PRIME 5

```
0900 DEF FNP(U)
0910 FOR D=3 TO SQR(U)
0920 Q=U/D
0930 IF INT(Q)=Q GOTO 0960
0940 NEXT D
0950 RETURN 1
0960 RETURN 0
0970 FNEND
1000 INPUT A,B
1010 IF INT(A/2)≠A/2 GOTO 1030
1020 A=A+1
1030 FOR N=A TO B STEP 2
1040 IF FNP(N)=0 GOTO 1060
1050 PRINT N;
1060 NEXT N
1070 PRINT
1080 END
```

L'instruction FNEND marque la fin de la définition de la fonction FNP; lors de l'exécution du programme, les lignes 900 à 970 sont ignorées par l'interpréteur tant qu'aucune instruc-tion ne fait appel à la fonction FNP

L'exécution de l'instruction 1040 déclenche par contre le calcul de FNP avec passage de la valeur de N au para-mètre U; le calcul de FNP s'achève lorsque l'interpréteur rencontre une instruction RETURN, et la valeur de l'expression qui suit RETURN est le résultat de l'évaluation de FNP.

Nous ne développerons pas plus l'étude de telles formes de sous - programmes, qui ne font pas partie du BASIC standard; l'avantage par rapport aux sous - programmes appelés par GOSUB est de permettre le 'passage d'arguments', par l'utilisation de paramètres qui sont traités indépendamment d'éventuelles variables homonymes du pro-gramme principal. Il reste un inconvénient source de nombreuses erreurs dans la mise au point de programmes complexes: les variables (autres que les paramètres) utilisées dans la définition de fonction doivent porter des noms distincts de celles utilisées dans le programme principal, sous peine de confusion. On arrive ainsi au résultat paradoxal suivant: une fonction définie correctement peut avoir des 'effets de bord' désastreux à l'intérieur de certains programmes, en modifiant de façon imprévue les valeurs de certaines variables 'principales'.

Bref, quelles que soient les améliorations qu'on puisse trouver dans certaines versions 'avancées' de BASIC, celui-ci reste un langage qui souffre gravement de ne pas avoir été conçu pour la collaboration entre procédures indépendantes.

[†] Cf. section 6.1

7 CHAINES DE CARACTERES

Un ordinateur n'est pas fait seulement pour traiter de l'information numérique. On peut l'utiliser par exemple à stocker des informations sur des personnes, à commencer par leurs noms. Par exemple, l'ordinateur de l'IREM de Nantes enregistre la candidature d'un stagiaire sous forme d'une chaine de caractères:

```
MR LAJOIE ALFRED     48TNANTES     44112473002N91N50N60N12N50*
```

(nous avons seulement changé le nom). Un ordinateur averti comprend que Monsieur Alfred LAJOIE est né en 1948, enseigne,dans un C.E.T. de Nantes (département 44), les mathématiques comme matière principale, à raison de 12 heures par semaine; c'est un PEGC depuis 1973, stagiaire à l'IREM (groupe N91) en 1976-77,etc... (certains renseignements sont évidemment codés; par exemple la lettre T, entre '48' et 'NANTES', signifie C.E.T., etc...).

§7.1 Constantes et variables littérales

Pour traiter les informations non numériques, BASIC emploie des constantes et des variables littérales -par opposition aux constantes et variables numériques. Une constante littérale est, en BASIC, une chaine de caractères placée entre apostrophes; exemple: 'DUPONT'. Un caractère peut être soit une lettre, soit un chiffre, soit n'importe quel signe spécial du clavier, comme +, ou ?, ou un blanc (obtenu en appuyant sur la barre d'espacement).Nous avons déjà rencontré des constantes littérales lors de l'étude de l'instruction PRINT (cf. section 3.3). Les apostrophes servent à l'interpréteur pour faire la distinction entre par exemple 'A', qui désigne la chaine de caractères constituée du seul caractère A, et la variable numérique A.

Une variable littérale est une variable destinée à recevoir pour valeur une chaine de caractères (et non un nombre). Les noms autorisés en BASIC pour ces variables sont formés d'une lettre suivie du symbole $:

```
A$, B$, C$, ..... , Z$
```

Les variables littérales peuvent être employées dans les instructions INPUT, READ et PRINT; les constantes littérales peuvent être employées dans les instruc-tions DATA et PRINT, ou bien en réponse à une instruction INPUT. On peut mêler expressions numériques et littérales, à condition de bien respecter les ordres

d'affectation: les variables numériques doivent recevoir des valeurs numériques, les variables littérales des valeurs littérales. Exemples:

```
0100 INPUT A$,A
0110 PRINT A$,A

RUN
?
'DUPONT',3
DUPONT              3

RUN
?
3,'DUPONT'

 ERROR 159  <-------  Les réponses n'ont pas été fournies dans
                      l'ordre convenable.
```

```
0100 DATA 'DUPONT',48,'DURAND',36
0110 READ A$,A
0120 READ B$,B
0130 PRINT A$,B$,A,B

RUN
DUPONT              DURAND              48                  36
```

Une variable littérale a une longueur fixée: sur IBM 5100 toute variable littérale comporte 18 caractères:

- si la valeur affectée à une variable littérale est une chaine de moins de 18 caractères, celle-ci est automatiquement complétée à droite par des blancs.
- si la valeur affectée est une chaine de plus de 18 caractères, celle-ci est automatiquement tronquée à droite.

Exemples:

```
0100 INPUT X$
0110 IF X$='FIN' GOTO 0140
0120 PRINT X$;
0130 GOTO 0100
0140 END

RUN
?
'DUPONT'
DUPONT
?
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ABCDEFGHIJKLMNOPQR
?
'123456789012345678901234'
123456789012345678
?
'ABC DEF GHI JKL MNP'
ABC DEF GHI JKL MN
```

Un blanc est un caractère comme un autre.

```
?
'J'EN AI MARRE'
J
?
'J''EN AI MARRE'
J'EN AI MARRE
?
'FIN'
```

Une apostrophe à l'intérieur d'une chaine de caractères doit être doublée en entrée; elle est alors traitée par l'interpréteur comme un caractère ordinaire (et non comme une délimita-tion de chaine).

Remarques.

1) Sur la plupart des systèmes, l'instruction PRINT ne provoque pas l'impression des blancs 'de remplissage'; exemple:

```
0100 DATA 'DUPONT',48,'DURAND',36
0110 READ A$,A
0120 READ B$,B
0130 PRINT A$,B$,A,B
0140 PRINT A$;B$
0150 PRINT A$;A;B$;B

RUN
DUPONT             DURAND             48
DUPONTDURAND
DUPONT 48   DURAND 36
```

2) Sur certains systèmes, et c'est beaucoup plus commode, la longueur d'une variable littérale peut être fixée par l'utilisateur, au moyen d'une instruction spéciale de déclaration (analogue à l'instruction DIM présentée section 5.1).

Instructions d'affectation.

Elles peuvent être simples, comme:

```
100 A$='DUPONT'
110 B$=A$
```

On peut aussi construire des expressions, grâce à des opérateurs applicables aux constantes et variables littérales. Les opérateurs numériques sont évidemment proscrits; une instruction comme:

```
A$=B$+C$
```

n'a pas de sens et est refusée par l'interpréteur. Les opérateurs littéraux dépendent beaucoup des systèmes. Le plus important, et parfois le seul présent, est l'opérateur appelé usuellement STR (abréviation de string), qui extrait une sous-chaine d'une chaine de caractères:

```
STR(V$,N,P)
```

est la chaine de P caractères, obtenue à partir de la variable V$ en extrayant P caractères à partir du $N^{ème}$; N et P peuvent être remplacées par des expressions numériques à valeurs entières; exemples:

```
STR(X$,A+B,2*X+1)

STR(F$,3,D+1)
```

Pour permettre à l'utilisateur de modifier certains caractères d'une chaine sans les modifier tous, une expression STR(.....) peut être placée à gauche du signe d'affectation =; exemple:

```
0100 A$='1234567890123456789'
0110 PRINT A$
0120 STR(A$,1,3)='ABC'
0130 PRINT A$
0140 B$='+x*?!/()=≤≠'
0150 STR(A$,14,5)=B$
0160 PRINT A$
0170 STR(A$,10,1)=STR(B$,3,1)
0180 PRINT A$

RUN
123456789012345678
ABC456789012345678
ABC4567890123+x*?!
ABC456789*123+x*?!
```

L'instruction 150 remplace les 5 derniers caractères de A$ par les cinq premiers caractères de B$; l'instruction 170 remplace le dixième caractère de A$ par le troisième de B$.

Les autres fonctions que l'on trouve sur certains systèmes sont:

- longueur: LEN(V$) désigne en général la longueur de V$ sans compter les blancs de remplissage de droite.
- concaténation: c'est l'opérateur qui permet de mettre bout à bout deux chaines de caractères.
- indice: permet de trouver, dans une chaine de caractères, la position d'un caractère donné (ou d'une sous-chaine).
- conversion variable littérale → variable numérique et vice-versa. En effet on a souvent besoin dans le même programme de considérer certaines valeurs tantôt comme numériques (pour faire des moyennes par exemple), tantôt comme littérales (pour les intégrer à une chaine de caractères).
- etc.....

Comparaisons.

Les constantes ou variables littérales peuvent être comparées et employées dans une instruction IF.... GOTO...; l'ordre est l'ordre alphabétique,avec les conventions supplémentaires suivantes: une lettre est inférieure à un chiffre, et un blanc est inférieur à une lettre; pour les autres caractères, consultez le manuel de référence de votre système.

Exemple:

```
0010 DATA 'DUPONT','DUPOND','ABC','ABCD'
0020 DATA 'X1','X2','1','A'
0030 DATA 'FIN'
0100 READ X$
0110 IF X$='FIN' GOTO 0210
0120 READ Y$
0130 IF X$<Y$ GOTO 0170
0140 IF X$>Y$ GOTO 0190
0150 PRINT X$;' = ';Y$
0160 GOTO 0100
0170 PRINT X$;' < ';Y$
0180 GOTO 0100
0190 PRINT X$;' > ';Y$
0200 GOTO 0100
0210 PRINT 'AU REVOIR'
0220 END

RUN
DUPONT > DUPOND
ABC < ABCD
X1 < X2
1 > A
AU REVOIR
```

Tableaux.

Les variables littérales peuvent former des tableaux, comme les variables numériques. La dimension, ou les dimensions, du tableau doivent être déclarées dans une instruction DIM (cf. section 5.1).

La section suivante montre un cas simple, mais assez typique, d'utilisation d'un tableau de variables littérales pour stocker et gérer un fichier de renseignements.

§7.2 Un exemple de 'gestion'

On suppose qu'on veut stocker et gérer des candidatures à l'IREM ; pour simplifier à l'extrême, nous supposerons qu'une candidature comporte seulement le nom du candidat et le code du groupe qu'il demande à suivre.

Ces renseignements constitueront un tableau littéral A$ à une dimension. Chaque élément A$(I) correspond à une candidature : les 15 premiers caractères sont réservés pour le nom du candidat, et les trois derniers pour le code du groupe.
Le programme:

- insère chaque nouveau candidat à sa place dans l'ordre alphabétique (lignes 120 à 190)
- fournit, une fois les candidats entrés, des listes de groupes, sur demande de l'utilisateur (lignes 200 à 310).

Le programme utilise deux sous-programmes ; le voici sans les détails des sous-programmes.

Programme STAGIREM

```
0099 REM PROGRAMME STAGIREM
0100 DIM A$(50)
0110 N=0
0120 PRINT 'ENTRER UN NOM, OU ''FIN''';
0130 INPUT R$
0140 IF R$='FIN' GOTO 0200
0150 PRINT 'GROUPE';
0160 INPUT G$
0165 STR(R$,16,3)=G$
0170 GOSUB 0500
0180 N=N+1
0190 GOTO 0120
0200 PRINT 'DESIREZ-VOUS LA LISTE D''UN GROUPE';
0210 INPUT R$
0220 IF R$='NON' GOTO 0300
0230 IF R$='OUI' GOTO 0260
0240 PRINT 'REPONDRE PAR OUI OU NON; JE REPETE:'
0250 GOTO 0200
0260 PRINT 'LEQUEL';
0270 INPUT G$
0280 GOSUB 0700
0290 GOTO 0200
0300 PRINT 'AU REVOIR!'
0310 END
0500 REM SOUS-PROGRAMME POUR INSERER R$ PARMI A$(1),...,A$(N)
0600 RETURN
0700 REM SOUS-PROGRAMME:LISTE DU GROUPE G$
0750 RETURN
```

Le programme principal est donc essentiellement un programme de dialogue avec l'utilisateur. Les constructions des lignes 120-140, et 200-250 sont employées fréquemment.

Le sous-programme qui débute à la ligne 700 ne présente pas de difficulté :

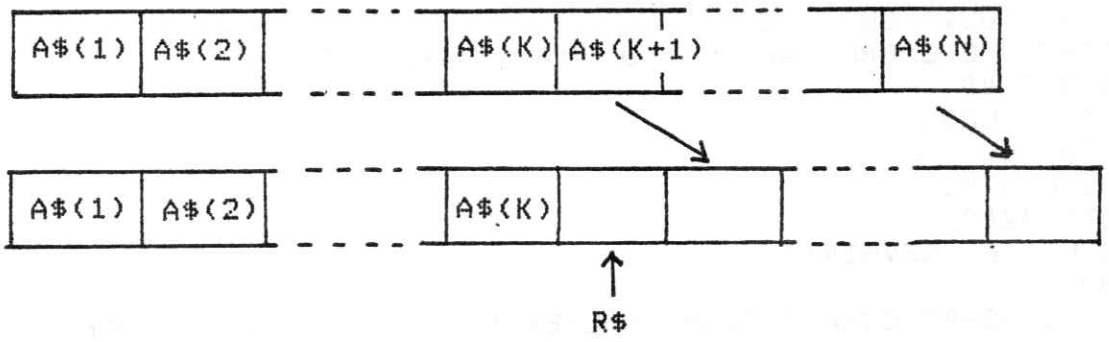
```
0700 REM SOUS-PROGRAMME:LISTE DU GROUPE G$
0705 PRINT
0706 PRINT
0710 FOR I=1 TO N
0720 IF G$≠STR(A$(I),16,3) GOTO 0740
0730 PRINT STR(A$(I),1,15)
0740 NEXT I
0745 PRINT
0750 RETURN
```

Le seul sous-programme qui demande un peu de réflexion est le sous-programme d'insertion de R$ parmi A$(1) , ... , A$(N), dans l'ordre alphabétique. Par construction A$(1), ... , A$(N) sont déjà rangés dans l'ordre alphabétique.

Il suffit donc de décaler vers la droite tous les A$(K) supérieurs à R$; on arrête dès qu'on a trouvé un A$(K) inférieur à R$, ou dès qu'on est arrivé au bout de la liste. Plus précisément, la procédure commence par :

K ⟵ N ;
Tant que K > 0 et R$<A$(K) faire
début 'Décaler à droite A$(K)' ; K ← K - 1 fin;

En fin de boucle, on a effectué les décalages suivants :



et la bonne place pour R$ est celle indiquée. Il ne reste donc qu'à faire :

A$(K+1)←R$

pour compléter l'insertion.

D'où le sous-programme BASIC :

```
0500 REM SOUS-PROGRAMME POUR INSERER R$ PARMI A$(1),...,A$(N)
0510 K=N
0520 REM BOUCLE TANT QUE K>0 ET R$<A$(K)
0530 IF K=0 GOTO 0590
0540 IF R$≥A$(K) GOTO 0590
0550 A$(K+1)=A$(K)
0570 K=K-1
0580 GOTO 0520
0590 A$(K+1)=R$
0600 RETURN
```

Exemple de fonctionnement

```
RUN
ENTRER UN NOM, OU 'FIN'
?
'MARTIN'
GROUPE
?
'N31'
ENTRER UN NOM, OU 'FIN'
?
'LE BOZEC'
GROUPE
?
'N32'
ENTRER UN NOM, OU 'FIN'
?
'BERTRAND'
GROUPE
?
'N31'
ENTRER UN NOM, OU 'FIN'
?
'DUPONT'
GROUPE
?
'N31'
ENTRER UN NOM, OU 'FIN'
?
'DURAND'
GROUPE
?
'N32'
ENTRER UN NOM, OU 'FIN'
?
'FIN'
DESIREZ-VOUS LA LISTE D'UN GROUPE
?
'OUI'
LEQUEL
?
'N32'

DURAND
LE BOZEC

DESIREZ-VOUS LA LISTE D'UN GROUPE
?
'BOF'
REPONDRE PAR OUI OU NON; JE REPETE:
DESIREZ-VOUS LA LISTE D'UN GROUPE
?
'OUI'
LEQUEL
?
'N31'

BERTRAND
DUPONT
MARTIN

DESIREZ-VOUS LA LISTE D'UN GROUPE
?
'NON'
AU REVOIR!
```

Remarques

1) Dans la réalité, on a besoin de beaucoup plus de renseignements par candidat. Comme sur IBM 5100 la longueur d'une variable littérale est fixée à 18 caractères, on doit tourner la difficulté en utilisant au moins un second tableau B$. Les renseignements relatifs au candidat numéro I sont alors stockés dans A$(I) et B$(I)
Cf. exercice.

2) Il faudrait aussi, évidemment, stocker le fichier obtenu sur bande magnétique (ou sur disque). Nous ne donnerons pas de précision sur la façon de procéder, très dépendante du système.

Exercice

1) Reprendre le programme précédent en supposant que chaque candidat choisit trois groupes (chacun codé sur trois caractères). On stockera les noms des candidats dans un tableau A$, et leurs voeux dans un tableau B$. On désire en outre que chaque liste de groupe commence par les candidats ayant choisi ce groupe en premier voeu, suivis des candidats l'ayant choisi en second voeu, enfin de ceux l'ayant choisi en troisième voeu.

§7.3 Permutations

Terminons ce chapitre par un sujet plus mathématique, l'étude d'un algorithme qui engendre toutes les permutations d'une chaine de caractères donnée α. Cet algo-rithme (il en existe probablement de meilleurs) utilise, pour engendrer le groupe des permutations d'une chaine de n caractères, les permutations circulaires suivantes:

t_1: permute les deux premiers éléments d'une chaine (t_1 est une transposition)
t_2: permute circulairement(vers la gauche) les 3 premiers éléments
t_3: " " 4 "

t_{n-1}: " " n "

On a le théorème suivant:

Toute permutation σ des n premiers caractères d'une chaine α peut être écrite de façon unique:

$$\sigma = t_1^{c_1} \circ t_2^{c_2} \circ \ldots \circ t_{n-1}^{c_{n-1}}$$

avec $\forall i: 0 \leq c_i \leq i$

La démonstration est simple, et l'idée qui la guide sera d'abord présentée sur un exemple. Soit σ la permutation 'ABCD' $\longmapsto$ 'ADBC', et cherchons à décomposer σ sous la forme:

$$\sigma = t_1^{c_1} \circ t_2^{c_2} \circ t_3^{c_3}$$

Comme seule la permutation circulaire t_3 agit sur le dernier élément de α, il faut choisir c_3 de telle sorte que:

$$t_3^{c_3}('ABCD') = '...C'$$

D'où: $c_3 = 3$ (rappelons que t_3 permute les éléments vers la gauche)

et : $$t_3^{c_3}('ABCD') = 'DABC'$$

On cherche ensuite c_2 tel que:

$$t_2^{c_2}('DABC') = '..BC'$$

d'où $c_2 = 0$, et c_1 doit être choisi tel que:

$$t_1^{c_1}('DABC') = 'ADBC'$$

D'où $c_1 = 1$

et: $$\sigma = t_1^1 \circ t_2^0 \circ t_3^3 = t_1 \circ t_3^3$$

La démarche suivie montre l'unité de cette décomposition.

Un raisonnement général peut se faire par induction:

- si n=2 le théorème est évident.
- dans le cas général, il existe une valeur et une seule de c_{n-1}, comprise entre 0 et n-1, telle que $t_{n-1}^{c_{n-1}}$ ait même effet que σ sur le $n^{ème}$ carac-

-tère de α . On a alors:

$\sigma = \sigma' \circ t_{n-1}^{c_{n-1}}$ où σ' désigne une permutation des (n-1) premiers caractères de α ; et donc, par hypothèse de récur-

-rence, σ' peut être écrite de façon unique:

$\sigma' = t_1^{c_1} \circ t_2^{c_2} \circ \ldots\ldots \circ t_{n-2}^{c_{n-2}}$ C.Q.F.D.

Ce théorème confirme d'ailleurs que le nombre de permutations de n éléments est:

$2 \times 3 \times \ldots\ldots \times n = n!$

(on a 2 choix pour c_1, 3 choix pour c_2, ..., n choix pour c_{n-1}).

L'idée de l'algorithme PERMUT (qui engendre et imprime les permutations des n premiers caractères d'une chaine donnée) est d'engendrer successivement les valeurs du vecteur C (à n-1 éléments):

$C = c_1\ c_2\ \ldots\ldots\ c_{n-1}$

dans l'ordre suivant (si par exemple n=4):

0	0	0
1	0	0
0	1	0
1	1	0
0	2	0
1	2	0
0	0	1
1	0	1
0	1	1
1	1	1
0	2	1
1	2	1
0	0	2
1	0	2
0	1	2
1	1	2
0	2	2
1	2	2
0	0	3
1	0	3
0	1	3
1	1	3
0	2	3
1	2	3

Liste des valeurs de

$C = c_1\ c_2\ c_3$

avec: $\forall i \quad 0 \leq c_i \leq i$

Cet ordre peut être appelé 'ordre lexicographique de droite à gauche'. Comment trouver par exemple le successeur de 1 2 1 3 ? Il suffit d'augmenter de 1 le premier élément, en partant de la gauche, qu'on puisse augmenter, et de mettre à zéro tous les éléments situés à sa gauche; on obtient: 0 0 2 3.

L'algorithme pour passer d'un vecteur C à son successeur est donc:

```
Q ← 1;
Tant que C(Q)=Q faire
        début  C(Q) ← 0; Q ← Q+1  fin;
C(Q) ← C(Q)+1;
```

Si cet algorithme est appliqué au dernier élément de la liste, l'exécution répétée de la boucle fournira :

Q = N

Cette condition servira de test de fin pour le programme PERMUT (pour ne pas avoir d'ennui avec le test C(Q)=Q dans ce cas, il vaudra mieux prévoir que le vecteur C comporte un $n^{ème}$ élément):

```
Départ  N ← n; C ← 0 0 0 ....0 (n éléments);
Répéter Q ← 1;
        tant que C(Q)=Q faire
            début C(Q) ← 0; Q ←Q+1  fin;
        si Q<N alors C(Q)← C(Q)+1
jusqu'à Q=N;
Fin
```

Cet algorithme engendre les valeurs successives de C dans l'ordre désiré; pour le vérifier, voici une traduction BASIC:

```
0100 DIM C(11)
0110 INPUT N
0120 MAT C=(0)
0130 GOSUB 0300
0140 REM BOUCLE JUSQU'A Q=N
0150 Q=1
0160 REM BOUCLE TANT QUE C(Q)=Q
0170 IF C(Q)<Q GOTO 0210
0180 C(Q)=0
0190 Q=Q+1
0200 GOTO 0160
0210 IF Q=N GOTO 0240
0220 C(Q)=C(Q)+1
0230 GOSUB 0300
0240 IF Q<N GOTO 0140
0250 END
0300 REM SOUS-PROGRAMME POUR IMPRIMER C
0310 FOR K=1 TO N-1
0320 PRINT C(K);
0330 NEXT K
0340 PRINT
0350 RETURN
```

Remarque: en BASIC il est évidemment préférable de remplacer les instructions 210 et 240 par:

```
210 IF Q=N GOTO 250
240 GOTO 140
```

C'est ce programme qui a fourni, pour n=4, la liste de la page précédente.

La clef de l'algorithme PERMUT réside maintenant dans le fait que de légères modifications de l'algorithme précédent permettent, en même temps qu'on passe d'un vecteur C à son suivant C', de passer de $t_C(\alpha)$ à $t_{C'}(\alpha)$, où α désigne une chaine de caractères, et:

$$t_C = t_1^{C(1)} \circ t_2^{C(2)} \circ \ldots\ldots \circ t_{n-1}^{C(n-1)}$$

Voici la partie centrale de l'algorithme (c.à.d. celle située à l'intérieur de la boucle Répéter ..... jusqu'à Q=N), après ces modifications, que nous justifierons ensuite:

```
Q ← 1;
tant que C(Q)=Q  faire      [A]
        début  A$ ← t_Q(A$);C(Q) ← 0;Q ← Q+1  fin;
si Q < N alors
        début  A$ ← t_Q(A$);C(Q) ← C(Q)+1  fin
```

Le lecteur peut simuler l'exécution de cet algorithme, avec au départ

$C = 1\ 2\ 1\ 3$ et $A\$ = t_C(\alpha) = t_1 \circ t_2^2 \circ t_3 \circ t_4^3\ (\alpha)$

pour bien en comprendre le fonctionnement. Donnons une preuve de correction en indi-quant les assertions qui restent valables à chaque passage au point [A]; ce sont:

(i) $J < Q \Rightarrow C(J) = 0$

(ii) $A\$ = t_C(\alpha) = t_Q^{C(Q)} \circ \ldots\ldots \circ t_{N-1}^{C(N-1)}\ (\alpha)$

Démonstration

- Au premier passage en [A], (i) est vraie car Q = 1, et (ii) est vraie par hypothèse.
- Montrons que si les assertions (i) et (ii) sont vraies avant exécution de la boucle, elles le restent après. L'instruction:

  $A\$ \leftarrow t_Q(A\$)$

  entraine:

  $A\$ = t_Q \circ t_Q^{C(Q)} \circ \ldots\ldots \circ t_{N-1}^{C(N-1)}(\alpha)$

  Comme la boucle n'est exécutée que si C(Q) = Q, on a donc:

  $A\$ = t_Q^{Q+1} \circ t_{Q+1}^{C(Q+1)} \circ \ldots\ldots \circ t_{N-1}^{C(N-1)}\ (\alpha)$

(3) $A\$ = t_{Q+1}^{C(Q+1)} \circ \ldots\ldots \circ t_{N-1}^{C(N-1)}\ (\alpha)$

  car $t_Q^{Q+1} = Id$ (rappelons que t_Q permute circulairement les $Q+1$ premiers éléments de α).

L'instruction $C(Q) \leftarrow 0$ entraine, vu (i):

(4) $\quad J \leq Q \Rightarrow C(J) = 0$

L'instruction $Q \leftarrow Q+1$ transforme alors (4) en (i) et (3) en (ii).

C.Q.F.D.

Quand on sort de la boucle tant que $C(Q)=Q$ faire .... on a donc:

(ii) $\quad A\$ = t_Q^{C(Q)} \circ \ldots \circ t_{N-1}^{C(N-1)} (\alpha) = t_C (\alpha)$

et: $\quad C(Q) < Q$

L'instruction: $\quad A\$ \leftarrow t_Q(A\$)$ entraine:

(5) $\quad A\$ = t_Q^{C(Q)+1} \circ \ldots \circ t_{N-1}^{C(N-1)} (\alpha)$

et l'instruction :

$C(Q) \leftarrow C(Q) + 1$ ramène (5) à (ii), ce qui prouve qu'en fin d'algo-rithme A$ a bien la valeur désirée.

Nous sommes désormais en mesure d'écrire l'algorithme définitif PERMUT , qui engendre et imprime les permutations des n premiers caractères d'une chaine α:

```
Départ  A$ ← α ; N ← n; C ← 0 0 0 ...0 (N termes); imprimer A$ ;
Répéter Q ← 1;
        tant que C(Q) = Q faire
                début A$ ← t_Q(A$); C(Q) ← 0; Q ← Q+1 fin;
        si Q < N alors
                début A$ ← t_Q(A$); C(Q) ← C(Q)+1; imprimer A$ fin
jusqu'à Q = N;
Fin†
```

Voici le programme BASIC correspondant:

---

† Rappelons, pour ceux qui prendraient le train en marche, que t_Q désigne la permutation circulaire vers la gauche des Q+1 premiers éléments d'une chaine.

```
0100 DIM C(11)
0105 PRINT 'CHAINE,N';
0110 INPUT A$,N
0120 MAT C=(0)
0130 PRINT A$;'     ';
0140 REM BOUCLE JUSQU'A Q=N
0150 Q=1
0160 REM BOUCLE TANT QUE C(Q)=Q
0170 IF C(Q)<Q GOTO 0210
0175 GOSUB 0400
0180 C(Q)=0
0190 Q=Q+1
0200 GOTO 0160
0210 IF Q=N GOTO 0250
0215 GOSUB 0400
0220 C(Q)=C(Q)+1
0230 PRINT A$;'     ';
0240 GOTO 0140
0250 END
0400 REM SOUS-PROGRAMME
0410 REM PERMUTATION CIRCULAIRE DES Q+1 PREMIERS ELEMENTS DE A$
0420 B$=STR(A$,1,1)
0430 STR(A$,1,Q)=STR(A$,2,Q)
0440 STR(A$,Q+1,1)=B$
0450 RETURN
```

Programme PERMUT

```
RUN
CHAINE,N
?
'ABCD',4
ABCD    BACD    BCAD    CBAD    CABD    ACBD    BCDA    CBDA
CDBA    DCBA    DBCA    BDCA    CDAB    DCAB    DACB    ADCB
ACDB    CADB    DABC    ADBC    ABDC    BADC    BDAC    DBAC
```

En remplaçant les instructions 130 et 230 par des appels à un petit sous-programme d'impression,on peut obtenir simultanément le vecteur C et la valeur de A$:

```
 0      0      0     ABCD
 1      0      0     BACD
 0      1      0     BCAD
 1      1      0     CBAD
 0      2      0     CABD
 1      2      0     ACBD
 0      0      1     BCDA
 1      0      1     CBDA
 0      1      1     CDBA
 1      1      1     DCBA
 0      2      1     DBCA
 1      2      1     BDCA
 0      0      2     CDAB
 1      0      2     DCAB
 0      1      2     DACB
 1      1      2     ADCB
 0      2      2     ACDB
 1      2      2     CADB
 0      0      3     DABC
 1      0      3     ADBC
 0      1      3     ABDC
 1      1      3     BADC
 0      2      3     BDAC
 1      2      3     DBAC
```

Remarques.

1) La structure d'un algorithme donné peut souvent être décrite de plusieurs façons. Voici une autre version du programme PERMUT, un peu plus courte, avec une seule boucle, et une alternative si ... alors ... sinon ... à l'intérieur de cette boucle:

```
0100 DIM C(10)
0110 INPUT A$,N
0120 Q=1
0130 MAT C=(0)
0140 PRINT A$;'     ';
0150 REM BOUCLE TANT QUE Q<N
0160 IF Q=N GOTO 0280
0170 B$=STR(A$,1,1)
0180 STR(A$,1,Q)=STR(A$,2,Q)
0190 STR(A$,Q+1,1)=B$
0200 IF C(Q)=Q GOTO 0250
0210 C(Q)=C(Q)+1
0220 PRINT A$;'     ';
0230 Q=1
0240 GOTO 0150
0250 C(Q)=0
0260 Q=Q+1
0270 GOTO 0150
0280 END
```

Ce programme fonctionne pratiquement exactement comme le précédent.

2) Voir page suivante ce qu'on obtient avec une chaine de 6 caractères. Durée d'exécution sur IBM 5100: environ 5 minutes. En extrapolant, on obtient:

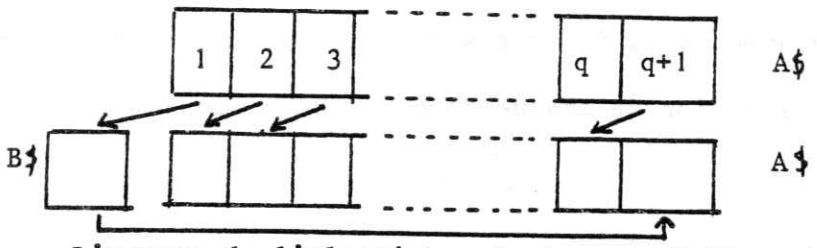
Longueur de la chaine	Nombre de pages pour écrire les permutations	Temps approximatif d'exécution
6	1	5 minutes
7	7	35 minutes
8	56	5 heures
9	504	2 jours
10	5040	20 jours

En informatique, on prend souvent 10! comme ordre de grandeur d'une ligne de partage entre le nombre de calculs qu'un ordinateur moderne peut, ou ne peut pas, exécuter en un temps raisonnable. On voit que sur un ordinateur portable construit autour d'un microprocesseur, la limite raisonnable est plutot 8!, avec évidemment toute l'imprécision du terme 'calcul', et sans oublier l'évolution rapide de la technologie des microprocesseurs.

```
RUN
?
'BRETON', 6
BRETON  RBETON  REBTON  ERBTON  EBRTON  BERTON  RETBON  ERTBON  ETRBON  TERBON  TREBON  RTEBON  ETBRON  TEBRON  TBERON  BTERON
BETRON  EBTRON  TBREON  BTREON  BRTEON  RBTEON  RTBEON  TRBEON  RETOBN  ERTOBN  ETROBN  TEROBN  TREOBN  RTEOBN  ETORBN  TEORBN
TOERBN  OTERBN  OETRBN  EOTRBN  TOREBN  OTREBN  ORTEBN  ROTEBN  RTOEBN  TROEBN  ORETBN  ROETBN  REOTBN  EROTBN  EORTBN  OERTBN
ETOBRN  TEOBRN  TOEBRN  OTEBRN  OETBRN  EOTBRN  TOBERN  OTBERN  OBTERN  BOTERN  BTOERN  TBOERN  OBETRN  BOETRN  BEOTRN  EBOTRN
EOBTRN  OEBTRN  BETORN  EBTORN  ETBORN  TEBORN  TBEORN  BTEORN  TOBREN  OTBREN  OBTREN  BOTREN  BTOREN  TBOREN  OBRTEN  BORTEN
BROTEN  RBOTEN  ROBTEN  ORBTEN  BRTOEN  RBTOEN  RTBOEN  TRBOEN  TBROEN  BTROEN  RTOBEN  TROBEN  TORBEN  OTRBEN  ORTBEN  ROTBEN
OBRETN  BORETN  BROETN  RBOETN  ROBETN  ORBETN  BREOTN  RBEOTN  REBOTN  ERBOTN  EBROTN  BEROTN  REOBTN  EROBTN  EORBTN  OERBTN
OREBTN  ROEBTN  EOBRTN  OEBRTN  OBERTN  BOERTN  BEORTN  EBORTN  RETONB  ERTONB  ETRONB  TERONB  TREONB  RTEONB  ETORNB  TEORNB
TOERNB  OTERNB  OETRNB  EOTRNB  TORENB  OTRENB  ORTENB  ROTENB  RTOENB  TROENB  ORETNB  ROETNB  REOTNB  EROTNB  EORTNB  OERTNB
ETONRB  TEONRB  TOENRB  OTENRB  OETNRB  EOTNRB  TONERB  OTNERB  ONTERB  NOTERB  NTOERB  TNOERB  ONETRB  NOETRB  NEOTRB  ENOTRB
EONTRB  OENTRB  NETORB  ENTORB  ETNORB  TENORB  TNEORB  NTEORB  TONREB  OTNREB  ONTREB  NOTREB  NTOREB  TNOREB  ONRTEB  NORTEB
NROTEB  RNOTEB  RONTEB  ORNTEB  NRTOEB  RNTOEB  RTNOEB  TRNOEB  TNROEB  NTROEB  RTONEB  TRONEB  TORNEB  OTRNEB  ORTNEB  ROTNEB
ONRETB  NORETB  NROETB  RNOETB  RONETB  ORNETB  NREOTB  RNEOTB  RENOTB  ERNOTB  ENROTB  NEROTB  REONTB  ERONTB  EORNTB  OERNTB
ORENTB  ROENTB  EONRTB  OENRTB  ONERTB  NOERTB  NEORTB  ENORTB  NRETOB  RNETOB  RENTOB  ERNTOB  ENRTOB  NERTOB  RETNOB  ERTNOB
ETRNOB  TERNOB  TRENOB  RTENOB  ETNROB  TENROB  TNEROB  NTEROB  NETROB  ENTROB  TNREOB  NTREOB  NRTEOB  RNTEOB  RTNEOB  TRNEOB
ETONBR  TEONBR  TOENBR  OTENBR  OETNBR  EOTNBR  TONEBR  OTNEBR  ONTEBR  NOTEBR  NTOEBR  TNOEBR  ONETBR  NOETBR  NEOTBR  ENOTBR
EONTBR  OENTBR  NETOBR  ENTOBR  ETNOBR  TENOBR  TNEOBR  NTEOBR  TONBER  OTNBER  ONTBER  NOTBER  NTOBER  TNOBER  ONBTER  NOBTER
NBOTER  BNOTER  BONTER  OBNTER  NBTOER  BNTOER  BTNOER  TBNOER  TNBOER  NTBOER  BTONER  TBONER  TOBNER  OTBNER  OBTNER  BOTNER
ONBETR  NOBETR  NBOETR  BNOETR  BONETR  OBNETR  NBEOTR  BNEOTR  BENOTR  EBNOTR  ENBOTR  NEBOTR  BEONTR  EBONTR  EOBNTR  OEBNTR
OBENTR  BOENTR  EONBTR  OENBTR  ONEBTR  NOEBTR  NEOBTR  ENOBTR  NBETOR  BNETOR  BENTOR  EBNTOR  ENBTOR  NEBTOR  BETNOR  EBTNOR
ETBNOR  TEBNOR  TBENOR  BTENOR  ETNBOR  TENBOR  TNEBOR  NTEBOR  NETBOR  ENTBOR  TNBEOR  NTBEOR  NBTEOR  BNTEOR  BTNEOR  TBNEOR
BETONR  EBTONR  ETBONR  TEBONR  TBEONR  BTEONR  ETOBNR  TEOBNR  TOEBNR  OTEBNR  OETBNR  EOTBNR  TOBENR  OTBENR  OBTENR  BOTENR
BTOENR  TBOENR  OBETNR  BOETNR  BEOTNR  EBOTNR  EOBTNR  OEBTNR  TONBRE  OTNBRE  ONTBRE  NOTBRE  NTOBRE  TNOBRE  ONBTRE  NOBTRE
NBOTRE  BNOTRE  BONTRE  OBNTRE  NBTORE  BNTORE  BTNORE  TBNORE  TNBORE  NTBORE  BTONRE  TBONRE  TOBNRE  OTBNRE  OBTNRE  BOTNRE
ONBRTE  NOBRTE  NBORTE  BNORTE  BONRTE  OBNRTE  NBROTE  BNROTE  BRNOTE  RBNOTE  RNBOTE  NRBOTE  BRONTE  RBONTE  ROBNTE  ORBNTE
OBRNTE  BORNTE  RONBTE  ORNBTE  ONRBTE  NORBTE  NROBTE  RNOBTE  NBRTOE  BNRTOE  BRNTOE  RBNTOE  RNBTOE  NRBTOE  BRTNOE  RBTNOE
RTBNOE  TRBNOE  TBRNOE  BTRNOE  RTNBOE  TRNBOE  TNRBOE  NTRBOE  NRTBOE  RNTBOE  TNBROE  NTBROE  NBTROE  BNTROE  BTNROE  TBNROE
BRTONE  RBTONE  RTBONE  TRBONE  TBRONE  BTRONE  RTOBNE  TROBNE  TORBNE  OTRBNE  ORTBNE  ROTBNE  TOBRNE  OTBRNE  OBTRNE  BOTRNE
BTORNE  TBORNE  OBRTNE  BORTNE  BROTNE  RBOTNE  ROBTNE  ORBTNE  RTONBE  TRONBE  TORNBE  OTRNBE  ORTNBE  ROTNBE  TONRBE  OTNRBE
ONTRBE  NOTRBE  NTORBE  TNORBE  ONRTBE  NORTBE  NROTBE  RNOTBE  RONTBE  ORNTBE  NRTOBE  RNTOBE  RTNOBE  TRNOBE  TNROBE  NTROBE
ONBRET  NOBRET  NBORET  BNORET  BONRET  OBNRET  NBROET  BNROET  BRNOET  RBNOET  RNBOET  NRBOET  BRONET  RBONET  ROBNET  ORBNET
OBRNET  BORNET  RONBET  ORNBET  ONRBET  NORBET  NROBET  RNOBET  NBREOT  BNREOT  BRNEOT  RBNEOT  RNBEOT  NRBEOT  BRENOT  RBENOT
REBNOT  ERBNOT  EBRNOT  BERNOT  RENBOT  ERNBOT  ENRBOT  NERBOT  NREBOT  RNEBOT  ENBROT  NEBROT  NBEROT  BNEROT  BENROT  EBNROT
BREONT  RBEONT  REBONT  ERBONT  EBRONT  BERONT  REOBNT  EROBNT  EORBNT  OERBNT  OREBNT  ROEBNT  EOBRNT  OEBRNT  OBERNT  BOERNT
BEORNT  EBORNT  OBRENT  BORENT  BROENT  RBOENT  ROBENT  ORBENT  REONBT  ERONBT  EORNBT  OERNBT  ORENBT  ROENBT  EONRBT  OENRBT
ONERBT  NOERBT  NEORBT  ENORBT  ONREBT  NOREBT  NROEBT  RNOEBT  RONEBT  ORNEBT  NREOBT  RNEOBT  RENOBT  ERNOBT  ENROBT  NEROBT
EONBRT  OENBRT  ONEBRT  NOEBRT  NEOBRT  ENOBRT  ONBERT  NOBERT  NBOERT  BNOERT  BONERT  OBNERT  NBEORT  BNEORT  BENORT  EBNORT
ENBORT  NEBORT  BEONRT  EBONRT  EOBNRT  OEBNRT  OBENRT  BOENRT  NBRETO  BNRETO  BRNETO  RBNETO  RNBETO  NRBETO  BRENTO  RBENTO
REBNTO  ERBNTO  EBRNTO  BERNTO  RENBTO  ERNBTO  ENRBTO  NERBTO  NREBTO  RNEBTO  ENBRTO  NEBRTO  NBERTO  BNERTO  BENRTO  EBNRTO
BRETNO  RBETNO  REBTNO  ERBTNO  EBRTNO  BERTNO  RETBNO  ERTBNO  ETRBNO  TERBNO  TREBNO  RTEBNO  ETBRNO  TEBRNO  TBERNO  BTERNO
BETRNO  EBTRNO  TBRENO  BTRENO  BRTENO  RBTENO  RTBENO  TRBENO  RETNBO  ERTNBO  ETRNBO  TERNBO  TRENBO  RTENBO  ETNRBO  TENRBO
TNERBO  NTERBO  NETRBO  ENTRBO  TNREBO  NTREBO  NRTEBO  RNTEBO  RTNEBO  TRNEBO  NRETBO  RNETBO  RENTBO  ERNTBO  ENRTBO  NERTBO
ETNBRO  TENBRO  TNEBRO  NTEBRO  NETBRO  ENTBRO  TNBERO  NTBERO  NBTERO  BNTERO  BTNERO  TBNERO  NBETRO  BNETRO  BENTRO  EBNTRO
ENBTRO  NEBTRO  BETNRO  EBTNRO  ETBNRO  TEBNRO  TBENRO  BTENRO  TNBREO  NTBREO  NBTREO  BNTREO  BTNREO  TBNREO  NBRTEO  BNRTEO
BRNTEO  RBNTEO  RNBTEO  NRBTEO  BRTNEO  RBTNEO  RTBNEO  TRBNEO  TBRNEO  BTRNEO  RTNBEO  TRNBEO  TNRBEO  NTRBEO  NRTBEO  RNTBEO
```

Analyse d'efficacité.

Une analyse sommaire de l'efficacité d'un algorithme de permutation consiste à évaluer le nombre de déplacements de caractères opérés en cours d'exécution. Or la permutation circulaire t_q, qui permute les q+1 premiers caractères d'une chaine, exige q+2 déplacements, suivant le schéma:



L'examen de l'algorithme de la page 7.15 montre que la boucle externe Répéter ... jusqu'à Q = N est exécutée n! fois, et que:

- une permutation t_1 est exécutée à chaque tour
- une permutation t_2 est exécutée un tour sur deux
- une permutation t_3 est exécutée un tour sur 6=3!
- etc.....

Le nombre de déplacements de caractères est donc donné par la formule:

$$M(n) = n!\ (3 + \frac{4}{2!} + \frac{5}{3!} + ... + \frac{n+1}{(n-1)!}\ )$$

$$= n!\ F(n)$$

On a le tableau suivant:

n	F(n)	M(n)
2	3.000	6
3	5.000	30
4	5.833	140
5	6.083	730
6	6.142	4422
7	6.153	31010
8	6.155	248152
9	6.155	2233458
10	6.155	22334690

On a : $F(n) = 2\ (1 + \frac{1}{2!} + \frac{1}{3!} + ... + \frac{1}{(n-1)!}$

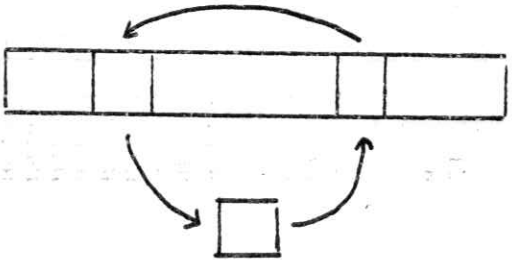
$$+ (1 + \frac{2}{2!} + \frac{3}{3!} + ... + \frac{n-1}{(n-1)!})$$

$$F(n) = 2\ (\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ... + \frac{1}{(n-1)!})$$

$$+ (1 + \frac{1}{1!} + \frac{1}{2!} + ... + \frac{1}{(n-2)!})$$

D'où $\lim F(n) = 2(e-1) + e = 3e - 2.$

Quel est le nombre de déplacements opéré par un algorithme optimum de ce point de vue? Le mieux serait évidemment de passer chaque fois d'une permutation à une autre par une transposition, qui exige 3 déplacements de caractères, suivant le schéma:



(On ne peut éviter un stockage intermédiaire)

Un algorithme optimum exige donc 3 n! déplacement de caractères ; l'algorithme PERMUT en utilise à peu près le double. Les exercices ci-dessous indiquent des algorithmes plus rapides. L'analyse complète de l'efficacité d'un programme doit évidemment tenir compte de l'ensemble du programme, mais il faut alors disposer de coefficients permettant d'évaluer la vitesse d'exécution de chaque type d'instruction.

Exercices

2) Le but est de construire un algorithme qui engendre la suite des permutations d'une chaîne de caractères, chacune étant déduite de la précédente par une simple transposition.

Comme dans l'algorithme PERMUT, on commence par associer à chaque permutation d'une chaîne de n caractères un vecteur C à n éléments, mais la signification de C est différente :

Si p est la permutation :

$$\alpha = \text{'ABCD ...'} \longrightarrow \beta$$

C(i) est le nombre de caractères situés à gauche de α_i dans la chaîne β, et supérieurs à α_i. Par exemple, avec :

$$p : \alpha = \text{'ABCDE'} \longrightarrow \text{'CBEDA'} = \beta$$

on a : C = 4 1 0 1 0

Car dans la chaîne β, il y a 4 caractères à gauche de 'A' et supérieurs à 'A'

1 ——————— 'B' ——————— 'B'

0 ——————— 'C' ——————— 'C'

1 ——————— 'D' ——————— 'D'

0 ——————— 'E' ——————— 'E'

On voit que

(1) $\forall i : 0 \leq C(i) \leq n - i$

En particulier $C(n)$ est toujours nul.

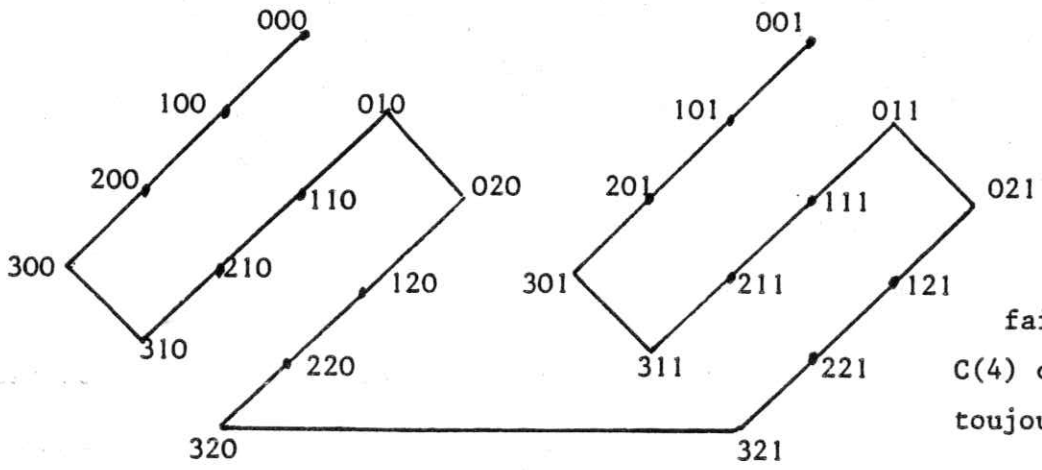
a) Montrer qu'inversement, on peut reconstruire la permutation p à partir du vecteur C

b) Montrer que si t est une <u>transposition élémentaire</u> (c'est-à-dire un échange de deux éléments consécutifs), et que si C (rest. C') est associé à p (resp. $t \circ p$), on a :

(2) $$\exists i : \begin{cases} C'(i) = C(i) \pm 1 \\ \text{et } j \neq i \Longrightarrow C'(j) = C(j) \end{cases}$$

c) Ecrire un algorithme qui engendre la suite des $n!$ valeurs possibles de C dans un ordre tel que la relation (2) soit toujours vérifiée entre un vecteur et son successeur.

<u>Indication</u> : se guider sur le dessin suivant, par exemple pour $n = 4$.



On n'a pas fait figurer $C(4)$ qui est toujours nul.

d) Compléter l'algorithme précédent afin d'engendrer la suite des permutations d'une chaîne donnée de n caractères.

3) Permutations dans l'ordre lexicographique.

Ecrire un algorithme SUCCESSEUR qui, à partir d'une chaîne α, génère la chaîne telle que β soit, parmi les chaînes qu'on peut déduire par permutation de α, la première dans l'ordre lexicographique après α. Certains caractères de α peuvent êt identiques.

Voici par exemple ce qu'on doit obtenir par itération de cet algorithme :

BETREMA

BETRMAE

BETRMEA

BMAEERT

Etc.....

BIBLIOGRAPHIE

Ouvrages de référence sur la programmation :

WIRTH Niklaus

[1] Systematic programming : an introduction (Prentice Hall)

[2] Algorithms + data structures = programs (Prentice Hall)

WIRTH est l'un des grands promoteurs de la "programmation structurée". Sur des exemples nombreux et intéressants (surtout dans [2]), il en expose les concepts de base (schémas de programmes, développement par raffinements successifs, preuves de programmes ...), en utilisant un langage développé ad hoc : PASCAL. [2] est important aussi pour l'étude des structures de données.

MANNA Zohar

[3] Mathematical theory of computation (Mac graw-Hill)

Ouvrage théorique de base sur les fonctions calculables, schémas de programmes, preuves de correction, schémas récursifs.

KNUTH D.E The art of computer programming (Addison-Wesley)

[4] Volume 1 : Fundamental algorithms

[5] Volume 2 : Semi-numerical algorithms

[6] Volume 3 : Sorting and searching

Une bible d'une grande clarté et d'une grande érudition. Les programmes présentés utilisent un langage d'assemblage fictif, MIX, mais il n'est pas nécessaire d'assimiler MIX pour utiliser ces ouvrages : les algorithmes sont toujours auparavant présentés très clairement.

Le volume 1 présente les algorithmes de base "au niveau machine" : la partie sur les "arbres" est spécialement intéressante. Le volume 2 présente les algorithmes utilisés en arithmétique et en génération de nombres aléatoires. Le volume 3 traite les algorithmes de tri et les structures facilitant l'accès rapide à des données.

Tous les algorithmes sont soigneusement analysées du point de vue efficacité : ces analyses mènent à l'étude de nombreux problèmes mathématiques intéressants.

Le volume 2 de la présente brochure utilisera largement quelques-unes des analyses présentées par KNUTH.

Sur l'utilisation de l'informatique dans l'enseignement français :

INRDP Recherches pédagogiques

[7] N° 54 : Emploi de calculateurs programmables dans le second degré.

[8] N° 75 : Calculateurs programmables dans les collèges et les lycées

APMEP (brochure n° 20)

[9] Quelques apports de l'informatique à l'enseignement des mathématiques

Ces brochures sont des mines d'idées pour des activités informatiques en liaison avec l'enseignement des mathématiques.

SOLUTIONS DES EXERCICES

Chapitre 1

1) Oui. Si a < b, la première exécution de :

R ⟵ reste de la division de A par B
A ⟵ B
B ⟵ R

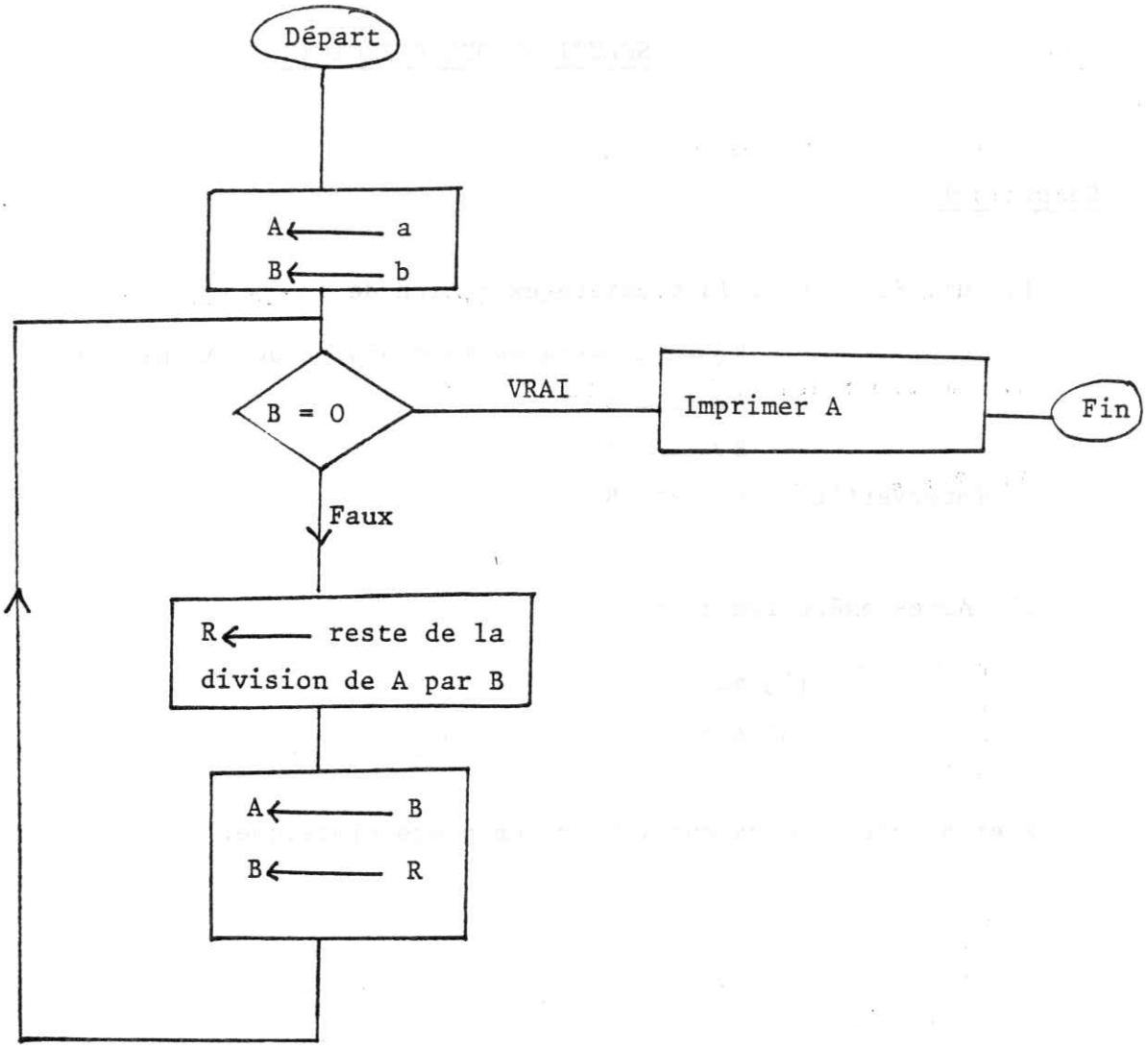
intervertit A et B

2) Après exécution de :

```
120 B=R
130 A=B
```

A et B ont même valeur ! C'est un piège classique.

3)



C'est l'organigramme 1.1, à ceci près qu'on a déplacé le test.

Chapitre 4

1) On obtiendra les valeurs de $\frac{1}{n}$ et de $\frac{1}{n^2}$; tous les calculs intermédiaires sont perdus.

2) Toutes les valeurs des sommes partielles intermédiaires sont imprimées.

3) Le programme FACT calcule la factorielle de la partie entière de n.

4) Par exemple :

```
0100 FOR K=1 TO 10
0110 PRINT K;
0120 NEXT K
0125 PRINT
0130 PRINT 'APRES EXECUTION DE LA BOUCLE, K=';K
0140 END
```

```
0100 N=0
0110 FOR K=10 TO 1
0120 N=N+1
0130 NEXT K
0140 PRINT 'BOUCLE EXECUTEE ';N;' FOIS!'
0150 END
```

5) Si le système exécute une fois la boucle 130-160, lorsque $3 > \sqrt{N}$ (avec par conséquent D=3) 1, 5 et 7 seront déclarés premiers par le programme, puisque non divisibles par 3, mais 3 ne sera pas retenu comme premier. On aura :

```
RUN
? 1,20
1 5 7 11 13 17 19
```

Si le système n'exécute pas la boucle 130-160 lorsque $3 \geq \sqrt{N}$, il y aura branchement direct en 170, et 1, 3, 5, 7 seront reconnus premiers.

```
RUN
? 1,20
1 3 5 7 11 13 17 19
```

(il ne manque alors que 2 pour que le programme soit parfait).

6) Surtout ne pas immerger le programme FACT dans une boucle du genre :

```
100 INPUT N1
105 FOR N=2 TO N1
110 |
    | inchangé
150 |
160 NEXT N1
```

Car il suffit, dans le programme initial, de permuter les instructions 140 et 150 !

7) Une solution est :

```
0100 REM CALCUL APPROCHE DE LA RACINE CARREE DE A
0110 INPUT A,E
0120 U=1
0130 REM BOUCLE JUSQU'A APPROXIMATION < E
0140 U=.5*(U+A/U)
0150 IF ABS(A-U↑2)>E GOTO 0130
0160 PRINT 'VALEUR APPROCHEE DE RACINE CARREE DE';A;
0165 PRINT 'A';E;'PRES:';U
0170 PRINT 'VALEUR DE LA FONCTION BASIC SQR:';SQR(A)
0180 END
```

8) Une solution est, si l'on suppose qu'on n'entre jamais de coefficient a nul, c'est-à-dire en écartant les équations dégénérées :

```
0100 PRINT 'COEFFICIENTS';
0110 INPUT A,B,C
0120 D=B↑2-4*A*C
0130 IF ABS(D)<1.E-06 GOTO 0170
0140 IF D>0 GOTO 0190
0150 PRINT 'PAS DE RACINE REELLE'
0160 STOP
0170 PRINT 'RACINE DOUBLE:';-B/(2*A)
0180 STOP
0190 D1=SQR(D)
0200 PRINT 'RACINES:';(-B+D1)/(2*A),(-B-D1)/(2*A)
```

9) Ce programme calcule C_n^k

```
0100 INPUT N,K
0110 C=1
0120 FOR J=1 TO K
0130 C=C*(N-J+1)/J
0140 NEXT J
0150 PRINT C
0160 END
```

Celui-ci calcule la $n^{\text{è}}$ ligne du triangle de Pascal :

```
0100 INPUT N
0110 C=1
0115 PRINT C;
0120 FOR J=1 TO N
0130 C=C*(N-J+1)/J
0135 PRINT C;
0140 NEXT J
0150 END
```

Après modification, on obtient le triangle de Pascal lui-même :

```
0100 INPUT N1
0105 FOR N=1 TO N1
0110 C=1
0115 PRINT C;
0120 FOR J=1 TO N
0130 C=C*(N-J+1)/J
0135 PRINT C;
0140 NEXT J
0145 PRINT
0146 NEXT N
0150 END
```

10) Démontrons que :

(i) Les assertions indiquées ci-dessous entre accolades sont vraies chaque fois qu'on passe par le point correspondant du programme lors de son exécution :

Départ $N \longleftarrow n$; $D \longleftarrow 2$; $z \longleftarrow$ vecteur vide ;
Tant que $D^2 \leqslant N$ faire
{(1) (D=2) ou (D impair $\geqslant 3$)
(2) $\forall i$: z_i premier
(3) $n = N \times \Pi z_i$
(4) ($Q \neq 1$ et Q divise N) $\Longrightarrow D \leqslant Q$}

si D divise N alors {(5) D premier}
début $z \longleftarrow z, D$; $N \longleftarrow N/D$ fin
sinon
si D=2 alors $D \longleftarrow 3$ sinon $D \longleftarrow D+2$;

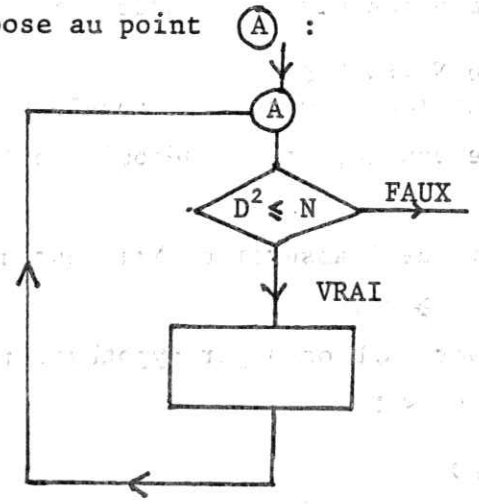
$z \longleftarrow z, N$ {(6) $n = \Pi z_i$ et $\forall i$: z_i premier}
Fin

(Conventions : z est le vecteur $z_1\ z_2 \ldots z_p$

Πz_i = si z est vide alors 1
sinon $z_1 . z_2 \ldots . z_p$

z, D désigne le vecteur $z_1\ z_2 \ldots z_p D$)

Précisons qu'il faut entendre que les assertions (1) - (2) - (3) - (4) sont vraies chaque fois qu'on pose au point (A) :



(ii) Le programme termine. C'est-à-dire : le prédicat ($D^2 \leqslant N$) devient faux après un nombre fini d'exécutions de la boucle :

Tant que $D^2 \leqslant N$ faire .....

Démonstration de (i) (preuve de correction partielle)

L'assertion (5) découle de (4) et de l'hypothèse : D divise N (D est le plus petit diviseur de N).

Les assertions (1) - (2) - (3) - (4) sont manifestement vraies avant d'entrer dans la boucle. Reste à vérifier, par induction, que si elles sont vraies avant exécution de la boucle, elles le restent après. C'est évident pour l'assertion (1) ; pour l'assertion (2), il suffit d'utiliser (5) ; (3) est manifestement invariante lorsqu'on exécute :

$$z \longleftarrow z, D \quad ; \quad N \longleftarrow N/D$$

Pour traiter (4), on distingue deux cas :

- si D divise N, alors N est remplacé par N/D, et comme tout diviseur Q de N/D divise aussi N, (4) reste vraie après exécution de la boucle.
- sinon, (4), qui est vraie par hypothèse de récurrence, entraine:

  (7) $(Q \neq 1$ et Q divise N$) \Rightarrow D+1 \leqslant Q$

  Si en outre $D \neq 2$, alors on a (cf. (1)):

  $$D \text{ impair} \geqslant 3$$

  donc (7) $\Rightarrow$ N impair , et D+1 ne divise pas N; donc dans ce cas:

  $$(Q \neq 1 \text{ et } Q \text{ divise } N) \Rightarrow D+2 \leqslant Q$$

  ce qui achève de prouver que (4) est un invariant de la boucle.

Enfin, en sortie de boucle, N n'a aucun diviseur inférieur à $\sqrt{N}$, puisque :

(4) $(Q \neq 1$ et Q divise N$) \Longrightarrow D \leqslant Q$
et $\sqrt{N} < D$ (condition de sortie de boucle).

Donc N est premier en sortie de boucle, et (6) découle de (2) et (3).

Remarque : on peut vérifier aussi que l'assertion $N \neq 1$ est un invariant du programme (à condition de prendre : $n \geqslant 2$).

En effet, quand on remplace N par N/D on a par hypothèse :

$$D^2 \leqslant N$$

d'où $2 \leqslant D \leqslant N/D$

Le programme ne fournit donc pas de facteur premier 'fictif' égal à 1.

Démonstration de (ii) (preuve de terminaison)

Il suffit de constater que l'expression :

$$N - D^2$$

décroit strictement à chaque exécution de la boucle.

Comme c'est une expression entière, après un nombre fini d'exécutions de la boucle on a :

$$N - D^2 < 0$$ (C.Q.F.D).

Chapitre 5

1) L'algorithme général est :

Départ N1 ⟵ n ; P(1) ⟵ 2 ; J ⟵ 1 ;
Pour N = 3 jusqu'à N1 pas 2 faire
si N premier alors début J ⟵ J+1 ; P(J) ⟵ N fin ;
Fin

Un programme possible est le suivant:

```
0100 DIM P(255)
0110 INPUT N1
0120 P(1)=2
0130 J=1
0140 FOR N=3 TO N1 STEP 2
0150 R=SQR(N)
0160 K=1
0170 REM BOUCLE TANT QUE P(K)≤R
0180 D=P(K)
0190 IF D>R GOTO 0250
0200 Q=N/D
0210 REM SORTIE ANORMALE DE BOUCLE SI D DIVISE N
0220 IF INT(Q)=Q GOTO 0270
0230 K=K+1
0240 GOTO 0170
0250 J=J+1
0260 P(J)=N
0270 NEXT N
0280 REM IMPRESSION DE P
0290 FOR I=1 TO J
0300 PRINT P(I);
0310 NEXT I
0320 PRINT
0330 END
```

Ce programme tourne correctement grâce au théorème suivant d'arithmétique (qui n'est pas du tout évident) :

si $p_1, p_2, \ldots p_j, \ldots$ désigne la suite des nombres premiers, on a :

$$\forall j : p_{j+1} < p_j^2$$

Ainsi la boucle 170-240 se termine toujours avec $K \leq J$

2) Une fois le vecteur P des nombres premiers inférieurs à N1 construit par le programme précédent, le programme suivant permet des décompositions en facteurs premiers :

```
0500 N2=N1↑2
0510 INPUT N
0520 IF N>N2 GOTO 0650
0530 K=1
0540 REM BOUCLE TANT QUE P(K)≤SQR(N)
0545 D=P(K)
0550 Q=N/D
0560 IF Q<D GOTO 0630
0570 IF INT(Q)=Q GOTO 0600
0580 K=K+1
0590 GOTO 0540
0600 PRINT P(K);
0610 N=Q
0620 GOTO 0540
0630 PRINT N
0640 GOTO 0510
0650 END
```

Chapitre 7

1) On peut par exemple ajouter la ligne :

```
105 DIM B$  (50)
```

effacer la ligne 165 du programme STAGIREM, ajouter un S à GROUPE dans l'instruction 150 (!) :

```
150 PRINT 'GROUPES' ;
```

et modifier les sous-programmes de la façon suivante :

```
0500 REM SOUS-PROGRAMME POUR INSERER R$ PARMI A$(1),...,A$(N)
0505 REM ET SIMULTANEMENT G$ PARMI B$(1),.....,B$(N)
0510 K=N
0520 REM BOUCLE TANT QUE K>0 ET R$<A$(K)
0530 IF K=0 GOTO 0590
0540 IF R$≥A$(K) GOTO 0590
0550 A$(K+1)=A$(K)
0560 B$(K+1)=B$(K)
0570 K=K-1
0580 GOTO 0520
0590 A$(K+1)=R$
0595 B$(K+1)=G$
0600 RETURN
0700 REM SOUS-PROGRAMME:LISTE DU GROUPE G$
0705 PRINT
0706 PRINT
0707 FOR J=1 TO 3
0708 PRINT 'VOEU NUMERO';J
0709 PRINT
0710 FOR I=1 TO N
0720 IF G$≠STR(B$(I),3*J-2,3) GOTO 0740
0730 PRINT A$(I),B$(I)
0740 NEXT I
0745 PRINT
0746 NEXT J
0750 RETURN
```

Remarque : en réponse à l'instruction :

160 INPUT G$

il faut entrer les 3 voeux sous forme d'une seule chaîne de 9 caractères.

2) Voici un programme, traduit en BASIC à partir d'un algorithme dû à TROTTER (1962), et présenté dans un article de ORD-SMITH (The Computer Journal : vol. 13 number 2. Mai 70 et vol. 14 number 2. Mai 71).

Dans cet algorithme plein d'astuce, le vecteur C (outre qu'il est renversé) est légèrement différent du vecteur C de l'énoncé de l'exercice, comme le montre l'exécutio qui suit. Le vecteur auxiliaire D aide à repérer les "sens de parcours" sur le chemin géométrique indiqué dans l'énoncé.

```
0100 DIM C(11),D(11)
0110 MAT C=(0)
0120 MAT D=(1)
0130 INPUT A$,N
0140 GOSUB 0500
0150 GOSUB 0300
0160 REM BOUCLE TANT QUE K≥2
0170 IF K=1 GOTO 0250
0180 J=Q+J
0190 T$=STR(A$,J,1)
0200 STR(A$,J,1)=STR(A$,J+1,1)
0210 STR(A$,J+1,1)=T$
0220 GOSUB 0500
0230 GOSUB 0300
0240 GOTO 0160
0250 END
0300 REM SOUS-PROGRAMME POUR MODIFIER C (ET D)
0310 K=N
0320 J=0
0330 REM BOUCLE JUSQU'A K=1
0340 REM SORTIE ANORMALE DES QUE 0<Q<K
0350 Q=C(K)+D(K)
0360 C(K)=Q
0370 IF Q=K GOTO 0410
0380 IF Q≠0 GOTO 0440
0390 REM J COMPTE LE NOMBRE DE PASSAGES PAR Q=0
0400 J=J+1
0410 D(K)=-D(K)
0420 K=K-1
0430 IF K≥2 GOTO 0330
0440 RETURN
0500 REM SOUS-PROGRAMME D'IMPRESSION
0510 FOR I=2 TO N
0520 PRINT FLP,C(I);
0530 NEXT I
0540 PRINT FLP,,
0550 FOR I=2 TO N
0560 PRINT FLP,D(I);
0570 NEXT I
0580 PRINT FLP,A$
0590 RETURN
```

Exemple d'exécution :

```
RUN
?
'ABCD',4

 0    0    0          1    1    1    ABCD
 0    0    1          1    1    1    BACD
 0    0    2          1    1    1    BCAD
 0    0    3          1    1    1    BCDA
 0    1    4          1    1   -1    CBDA
 0    1    3          1    1   -1    CBAD
 0    1    2          1    1   -1    CABD
 0    1    1          1    1   -1    ACBD
 0    2    0          1    1    1    ACDB
 0    2    1          1    1    1    CADB
 0    2    2          1    1    1    CDAB
 0    2    3          1    1    1    CDBA
 1    3    4          1   -1   -1    DCBA
 1    3    3          1   -1   -1    DCAB
 1    3    2          1   -1   -1    DACB
 1    3    1          1   -1   -1    ADCB
 1    2    0          1   -1    1    ADBC
 1    2    1          1   -1    1    DABC
 1    2    2          1   -1    1    DBAC
 1    2    3          1   -1    1    DBCA
 1    1    4          1   -1   -1    BDCA
 1    1    3          1   -1   -1    BDAC
 1    1    2          1   -1   -1    BADC
 1    1    1          1   -1   -1    ABDC
```

Evidemment le sous-programme qui débute à la ligne 500 n'imprime les vecteurs C et D que pour éclairer la lanterne du lecteur. L'instruction PRINT FLP signifie, sur IBM 5100 : imprimer ... sur l'imprimante (alors que PRINT correspond à un affichage sur écran).

Remarque : Le programme ci-dessus a été partiellement 'bien structuré' ; il y gagne, je l'espère, en clarté, mais perd un peu en efficacité par rapport à l'algorithme original de TROTTER.

3) Voici un algorithme pour passer d'une chaîne A$ de longueur N à la suivante dans l'ordre alphabétique, par permutation des caractères :

```
DEPART  ENTRER A$,N;
J←N-1;
TANT QUE J≠0 ET A$[J+1]≤A$[J] FAIRE J←J-1;
SI J≠0 ALORS DEBUT
             K←N;
             TANT QUE A$[K]≤A$[J] FAIRE K←K-1;
             ECHANGER A$[J] ET A$[K]
             FIN;
¤REMISE EN ORDRE DE A$[J+1,...,N]¤
I←J+1; J←N;
TANT QUE I<J FAIRE DEBUT
                   ECHANGER A$[I] ET A$[J];
                   I←I+1; J←J-1
                   FIN;
FIN
```

Voici un programme qui génère, dans l'ordre lexicographique, à partir d'une chaîne A$, toutes les chaînes qui s'en déduisent par permutation, jusqu'à une chaîne donnée Z$; il repose sur l'algorithme précédent, mais, pour aller plus vite, traite à part les permutations des 2 ou 3 derniers caractères (cf. article précité de ORD-SMITH):

Programme LEX:

```
0010 REM ORDRE LEXICOGRAPHIQUE
0020 PRINT 'CHAINE DEPART  , N , CHAINE ARRIVEE';
0030 INPUT A$,N,Z$
0040 GOSUB 0605
0050 GOSUB 0610
0060 REM BOUCLE JUSQU'A A$=Z$
0090 STR(B$,1,1)=STR(A$,N,1)
0110 STR(C$,1,1)=STR(A$,N-1,1)
0130 IF B$≤C$ GOTO 0180
0140 STR(A$,N-1,1)=B$
0150 STR(A$,N,1)=C$
0160 GOSUB 0610
0170 GOTO 0590
0180 STR(D$,1,1)=STR(A$,N-2,1)
0200 IF C$≤D$ GOTO 0320
0210 IF B$≤D$ GOTO 0270
0220 STR(A$,N,1)=C$
0230 STR(A$,N-1,1)=D$
0240 STR(A$,N-2,1)=B$
0250 GOSUB 0610
0260 GOTO 0590
0270 STR(A$,N-2,1)=C$
0280 STR(A$,N-1,1)=B$
0290 STR(A$,N,1)=D$
0300 GOSUB 0610
0310 GOTO 0590
0320 REM CAS GENERAL
0330 J=N-3
0340 REM BOUCLE TANT QUE J≠0 ET A$[J+1]≤A$[J]
0350 IF J=0 GOTO 0470
0360 IF STR(A$,J,1)<STR(A$,J+1,1) GOTO 0390
0370 J=J-1
0380 GOTO 0340
0390 K=N
0400 REM BOUCLE TANT QUE A$[K]≤A$[J]
0410 IF STR(A$,K,1)>STR(A$,J,1) GOTO 0440
0420 K=K-1
0430 GOTO 0400
0440 T$=STR(A$,J,1)
0450 STR(A$,J,1)=STR(A$,K,1)
0460 STR(A$,K,1)=T$
0470 REM REMISE EN ORDRE DE A$[J+1 ... N]
0480 I=J+1
0490 J=N
0500 REM BOUCLE TANT QUE I<J
0510 IF I≥J GOTO 0580
0520 T$=STR(A$,I,1)
0530 STR(A$,I,1)=STR(A$,J,1)
0540 STR(A$,J,1)=T$
0550 I=I+1
0560 J=J-1
0570 GOTO 0500
0580 GOSUB 0610
0590 IF A$≠Z$ GOTO 0060
0600 END
```

Les sous-programmes 605 et 610 servent seulement à l'impression des résultats; ils sont un peu sophistiqués, et liés aux caractéristiques de l'imprimante: ils permettent de limiter la largeur de chaque ligne à 80 caractères, et de sauter 6 lignes toutes les 60 lignes (!) (pour ne pas écrire sur les 'pliures' du papier en liasse....; chaque feuille comporte 66 lignes).

```
0605 REM SOUS-PROGRAMME: PREPARATION IMPRESSION
0606 C1=INT(80/(N+2))
0607 C=1
0608 D=1
0609 RETURN
0610 REM SOUS-PROGRAMME D'IMPRESSION
0620 PRINT FLP,A$;'   ';
0630 IF C=C1 GOTO 0660
0640 C=C+1
0650 RETURN
0660 PRINT FLP,
0670 C=1
0680 IF D=60 GOTO 0710
0690 D=D+1
0700 RETURN
0710 FOR I1=1 TO 6
0720 PRINT FLP,
0730 NEXT I1
0740 D=1
0750 RETURN
```

Exemple (pas trop long...) d'exécution:

```
RUN
CHAINE DEPART  , N , CHAINE ARRIVEE
?
'PIERRE',6,'PIERRE'

PIERRE  PIREER  PIRERE  PIRREE  PREEIR  PREERI  PREIER  PREIRE  PREREI  PRERIE
PRIEER  PRIERE  PRIREE  PRREEI  PRREIE  PRRIEE  REEIPR  REEIRP  REEPIR  REEPRI
REERIP  REERPI  REIEPR  REIERP  REIPER  REIPRE  REIREP  REIRPE  REPEIR  REPERI
REPIER  REPIRE  REPREI  REPRIE  REREIP  REREPI  RERIEP  RERIPE  RERPEI  RERPIE
RIEEPR  RIEERP  RIEPER  RIEPRE  RIEREP  RIERPE  RIPEER  RIPERE  RIPREE  RIREEP
RPIREE  RPREEI  RPREIE  RPRIEE  RREEIP  RREEPI  RREIEP  RREIPE  RREPEI  RREPIE
RRIEEP  RRIEPE  RRIPEE  RRPEEI  RRPEIE  RRPIEE  EEIPRR  EEIRPR  EEIRRP  EEPIRR
EIERRP  EIPERR  EIPRER  EIPRRE  EIREPR  EIRERP  EIRPER  EIRPRE  EIRREP  EIRRPE
EPEIRR  EPERIR  EPERRI  EPIERR  EPIRER  EPIRRE  EPREIR  EPRERI  EPRIER  EPRIRE
EPRREI  EPRRIE  EREIPR  EREIRP  EREPIR  EREPRI  ERERIP  ERERPI  ERIEPR  ERIERP
ERIPER  ERIPRE  ERIREP  ERIRPE  ERPEIR  ERPERI  ERPIER  ERPIRE  ERPREI  ERPRIE
ERREIP  ERREPI  ERRIEP  ERRIPE  ERRPEI  ERRPIE  IEEPRR  IEERPR  IEERRP  IEPERR
IEPRER  IEPRRE  IEREPR  IERERP  IERPER  IERPRE  IERREP  IERRPE  IPEERR  IPERER
IPERRE  IPREER  IPRERE  IPRREE  IREEPR  IREERP  IREPER  IREPRE  IREREP  IRERPE
IRPEER  IRPERE  IRPREE  IRREEP  IRREPE  IRRPEE  PEEIRR  PEERIR  PEERRI  PEIERR
PEIRER  PEIRRE  PEREIR  PERERI  PERIER  PERIRE  PERREI  PERRIE  PIEERR  PIERER
PIERRE
```