I.U.T. de Marne-La-Vallée

Introduction à l'informatique et programmation en langage C

(DUT Génie Thermique et Energie)

Jean Fruitet

Jean.Fruitet@univ-mlv.fr

1999

I.U.T. de Marne-La-Vallée

Introduction à l'informatique et programmation en langage C

(DUT Génie Thermique et Energie)

Jean Fruitet

Jean.Fruitet@univ-mlv.fr

Avertissement

Ce cours s'adresse aux étudiants de première année de DUT de Génie Thermique et Energie (GTE). Il leur est présenté en quelques dizaines d'heures —une trentaine— les rudiments de la programmation numérique et des notions d'algorithmique. Ces étudiants n'étant pas destinés à une carriére d'informaticien professionnel, je n'aborde pas l'algorithmique dans tous ses raffinements. En particulier les notions pourtant fondamentales de preuve de programme et d'analyse de complexité ne sont pas évoquées.

Ce cours est divisé en quatre parties :

- notion d'informatique et de codage ;

- structure d'un ordinateur : la machine à accès direct (MAD / RAM) ;

- langage de programmation : le langage C ;

- algorithmique numérique et structures de données.

Après quelques notions de théorie de l'information et de codage (codage binaire, représentation des entiers et des flottants) j'introduis la programmation de fonctions numériques sur ordinateur PC sous MS-DOS puis l'utilisation de quelques structures de données fondamentales (tableaux, piles, files, arbres, graphes) et les principaux algorithmes de tri. Ce cours ne fait donc aucune place à la technologie des ordinateurs, leur architecture, système d'exploitation et de fichiers. Il n'est pas non plus question d'apprentissage de logiciels bureautiques (traitement de texte ou de tableur). Ce n'est pas que ces connaissances ne soient pas nécessaires aux techniciens, mais je laisse à d'autres enseignants le soin d'y contribuer.

S'agissant de la syntaxe d'un langage de programmation, j'introduis le langage RAM, pour passer rapidement au langage C. J'insiste beaucoup dans ce cours sur la nécessité d'une programmation structurée descendante. Cette démarche est recommandée depuis des lustres par tous les spécialistes. Malheureusement l'expérience montre que livré à lui-même le programmeur moyen se permet des libertés qui rendent rapidement ses programmes illisibles et inutilisables. Mais ce ne sera pas faute d'avoir été prévenu...

Caractérisation d'un problème informatique

L'art de programmer, c'est l'art de faire résoudre des problèmes par des machines. Il s'agit bien d'un art, au sens de l'artisan, qui passe par une longue période d'apprentissage et d'imitation. Dans cet exercice certains individus ont des dispositions naturelles ; pour les autres un apprentissage rigoureux fournit les rudiments d'une méthode. Le reste est affaire de travail et d'investissement personnels.

Un ordinateur est dénué d'intelligence ; il ne peut donc résoudre que les problèmes pour lesquels existe une méthode de résolution algorithmique, c'est-à-dire une recette déterministe. De plus, même si la recette existe en théorie pour résoudre tel problème, encore faut-il que l'énoncé du problème — l'espace des paramètres— et l'ensemble des solutions soient de dimension finie, en raison de la limitation en taille de la mémoire des machines. Enfin, condition ultime, la mise en oeuvre d'un algorithme doit avoir une durée finie. Un problème dont la solution nécessite de disposer d'un temps infini n'est pas considéré comme résoluble par ordinateur. Ces trivialités vont donc limiter nos ambitions de programmeur à une classe de problèmes assez restreinte, d'autant que ne nous disposons pas de puissantes machines de calcul.

2.1. Le traitement de l'information

L'informatique est la science du traitement de l'information. Une information est un élément de connaissance susceptible d'être codé, transmis et traité de manière automatique. Le codage numérique en nombres binaires étant adapté à la conception de machines électroniques, une partie essentielle du cours d'informatique porte sur la représentation des nombres et la logique (algèbre de Boole) binaires. Je considèrerai comme acquises les notions d'opération élémentaire (addition, soustraction, multiplication et division réelle et euclidienne) sur les ensembles de nombres naturels, relatifs, rationnels, réels et complexes, qui ne seront pas redéfinies, non plus que les opérations ensemblistes union, intersection, complémentation, ni les notions de relation binaire, relation d'équivalence et relation d'ordre partiel ou total. Concernant les notions de constante numérique, de variable, d'instruction d'affectation, de test, de boucle, qui sont au centre des techniques de programmation, elles seront redéfinies ou précisées selon les besoins.

2.2. Quelques exemples de problèmes

L'ordinateur fonctionne en binaire, mais il peut résoudre des problèmes autres que numériques. Et bien que le calculateur électronique soit l'instrument privilégié de l'ingénieur, ce n'est pas en programmant des problèmes numériques qu'on apprend le plus efficacement à programmer. Pourtant il est de tradition dans les sections techniques et scientifiques de commencer par des exercices numériques :

- Résoudre une équation du second degré à coefficients réels dans le corps de nombres complexes.

- Programmer la division euclidienne de deux entiers en n'employant que des soustractions.

- Trouver le plus grand diviseur commun de deux nombres naturels (PGDC).

- Enumérer les *n* premiers nombres premiers.

- Tester la conjecture polonaise.

- Calculer le nième élément de la suite de Fibonacci.

- Calculer le minimum, le maximum, la moyenne et l'écart-type d'une distribution numérique.

- Calculer les zéros d'un polynome, tracer graphiquement le graphe d'une fonction numÉrique

- inverser une matrice.

D'autres problèmes qui ne sont pas strictement numériques sont pourtant tout aussi instructifs pour l'art de la programmation. Ils seront soit traités soit évoqués :

- Trouver un mot dans un dictionnaire.

- Construire un arbre hiérarchique

- Ordonner une liste de mots.

- Fusionner deux listes de mots déjà ordonnés.

- Enumérer les parcours d'un graphe et trouver le plus court chemin entre deux sommets.

- Filtrer une image, etc.

Démarche

Partant d'un problème élémentaire nous montrerons comment le reformuler en des termes susceptibles d'être traités par un ordinateur idéal. Puis nous coderons ces algorithmes en langage C. Nous encourageons le lecteur à implanter ses propres solutions, à les modifier, éventuellement les améliorer ou à les réutiliser pour d'autres applications.

Tous les exemples fournis en C ont été testés sur compilateur Turbo C sur PC et Think C sur Macintosh. Je remercie par avance celles et ceux qui voudront bien me transmettre remarques et suggestions.

Introduction à l'informatique

L'Informatique est la science du traitement automatique de l'information.

La notion d'information est assez générale. Pour l'informatique elle prend un sens particulier : *Une* information *est un élément ou un système de connaissance pouvant être transmis au moyen d'un* support *et d'un* codage *approprié et pouvant être compris*.

Par exemple des hiéroglyphes (codage) sur un papyrus égyptien (support) constituent une information sur la société égyptienne antique dés qu'on a été en mesure de les lire et d'en comprendre le sens (Champollion au XIXéme siècle).

Une information est une fonction du temps, puisque le contenu d'un message est sujet à changer au cours du temps. L'information "La mer est 'calme' dans le Golfe de Gascogne" est particulièrement périssable... Quand le vent se léve et que la houle se forme, la mer devient 'agitée'. L'état de la mer est une information qui peut prendre des valeurs différentes, au cours du temps.

Langage

La plupart des informations que les Humains échangent sont supportées par un langage, c'est-à-dire des groupes de sons (les mots), qu'il faut assembler "d'une certaine manière" (grammaire) pour que les phrases aient un sens... Avec l'invention de l'écriture, ces mots ont eu une transcription (recodage) sous forme de symboles graphiques (des formes) dessinés ou imprimés.

Le concept de mot est fondamental en informatique, de même que celui de langage. Un langage est un ensemble de mots construits avec les lettres choisies dans un *alphabet*. Les mots sont assemblés en phrases selon des régles de grammaire précises qui définissent la *syntaxe* du langage. Le sens attaché à ces phrases, c'est-à-dire leur signification, constitue la *sémantique*.

L'essentiel de ce cours va consister à expliquer comment sont commandées les machines que nous nommons ordinateurs, capables d'exécuter des tâches complexes de traitement de l'information.

Traitement de l'information

Le traitement de l'information consiste en une suite d'opérations transformant une représentation de cette information en une autre représentation plus facile à manipuler ou à interpréter.

Exemples :

"3*2" remplacé par "6"

"Mille neuf cent quatre vingt treize" est remplacé par "1993"

"La somme des carrés des côtés de l'angle droit d'un triangle rectangle est égale au carré de l'hypoténuse" est remplacé par "Théoréme de Pythagore"

"Championne olympique 1992 et 1996 du 400 mètres féminin" est remplacé par "Marie-José Pérec".

Dans une entreprise, traiter l'information peut consister à établir la paye, faire la facturation, gérer le stock, dresser un bilan. Dans un atelier, diriger un robot. En météorologie, reconnaître un cyclone sur une photo satellite...

Ordinateur

Un ordinateur est une machine qui permet d'effectuer des traitements sur des données à l'aide de programmes. Les données (les paramètres du problème, par exemple les notes des étudiants du cours d'informatique) ainsi que le programme (par exemple le calcul de la moyenne des notes) sont fournis à la machine par l'utilisateur au moyen de dispositifs de saisie (le clavier). Le résultat du traitement est recueilli à la sortie de l'ordinateur (l'écran, l'imprimante) sous forme de texte.

Un peu d'histoire

C'est en 1945 que le principe des ordinateurs a été inventé. Mais on peut faire remonter ses origines au boulier et aux premières machines à calculer mécaniques. Blaise Pascal (1623-1662) inventa à l'âge de 18 ans une machine à base de roues dentées et d'engrenages qui réalise d'elle-même les additions et les soustractions. Il suffit d'indiquer les chiffres et l'opération à faire.

Au XIXéme siècle l'anglais Babbage conçoit deux grandes machines dont le principe était correct, mais qui ne purent être réalisées en raison de difficultés techniques et financiéres. Il était sans doute trop tôt. Ce n'est qu'au XXème siècle que sous la pression du développement économique et des besoins militaires (Deuxiéme guerre mondiale) des scientifiques et des ingénieurs s'attelérent à la construction de gigantesques machines à calculer. La plus fameuse de ces machines fut la "Harvard Mark 1" qui mesurait 16 mètres de long, pesait 5 tonnes et comprenait 800 000 éléments, et pourtant n'avait pas plus de puissance qu'une simple calculette de poche actuelle !

La véritable révolution pour les machines à calculer viendra des progrès de l'électronique et de la logique mathématique. Le premier calculateur électronique, l'ENIAC, destiné à calculer la trajectoire de projectiles pour l'armée américaine, fut construit à partir de 1943. Il pesait 30 tonnes, comportait 17 468 tubes à vide et additionnait 5000 nombres en une seconde. Mais on ne peut considèrer cette machine comme un ordinateur, car il n'était pas véritablement automatique et n'utilisait pas de programme interne.[1]

Sur le plan technique les progrès décisifs seront réalisés dans les années 1950 avec l'invention en 1947 du transistor (qui donne son nom aux postes de radio portables). Les transistors sont des composants électroniques qui remplace partout les lampes à vides ; rassemblés par dizaines puis centaines de milliers sur des circuits intégrés ils permettent de réaliser des puces électroniques qui envahissent les automates (lave linge, magnétoscopes, circuits d'allumage de voiture, calculettes...) et les ordinateurs.

Sur le plan conceptuel, c'est aux anglais George Boole (1815-1864), inventeur de l'algèbre binaire, l'algèbre de la logique, et Alan Turing (1912-1954) et aux américains d'origine européenne John Von Neumann (1903-1957) et Norbert Wiener (1894-164) que nous devons l'avancée décisive qui méne des calculateurs aux ordinateurs. Leurs travaux aboutissent à la construction du premier ordinateur en 1948 à Manchester, en Grande-Bretagne, le "Manchester Mark 1".

Ce qui caractérise un ordinateur

La machine conçue par John Von Neumann comporte trois innovations majeures :

- elle a une mémoire importante, dans laquelle sont archivées les données et le programme.

- elle a un programme enregistré dans la mémoire, qui décrit l'ensemble des instructions à réaliser.

[1] Philippe BRETON "Une histoire de l'Informatique" - Collection Points Sciences - Editions La Découverte, Le Seuil 1990.

- elle a une unité centrale de commande interne qui organise le travail en appliquant les instructions du programme et dirige les échanges de données avec l'extérieur de la machine.

Matériel et logiciel

Un ordinateur est constitué de composants matériels (*hardware*) et de composants logiciels (*software*). Les composants matériels sont essentiellement des cartes électroniques, des circuits intégrés, des câbles électriques, des supports de mémoires de masse (disques durs) et des dispositifs d'entrée/sortie (périphériques : clavier, écran, imprimante). Les logiciels, qui pilotent le fonctionnement des composant matériels, sont des programmes stockés sous forme codée dans la mémoire de l'ordinateur. Pour être interprétés par l'unité centrale ces programmes doivent être traduits dans le langage des machines, le langage binaire.

Le codage binaire

Toute l'information qui transite dans un ordinateur est codée avec des mots formés seulement de deux symboles (ou de deux 'états') notés 0 et 1. Cela tient à la nature des composants électriques et magnétiques utilisés pour coder l'information.

Dans les mémoires d'ordinateur, chaque unité élémentaire d'information peut être représentée par un minuscule aimant. Chaque aimant est orienté soit dans un sens (état 0), soit dans le sens opposé (état 1)...

C'est le même principe qui est appliqué aux échanges de données. Pour transmettre un 1, il faut appliquer sur un conducteur électrique une différence de potentiel supérieure à quelques volts pendant une période "assez" longue, de l'ordre de la micro seconde (1/1 000 000 éme de seconde). Pour transmettre un 0, il faut maintenir une différence de potentiel inférieure à 1 volt pendant la même durée.

Par exemple pour coder le nombre 13 en binaire, il faut les quatre chiffres binaires **1101**. En effet 13 peut être décomposé comme une somme de puissances de 2

13 = 8 + 4 + 1
= 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 en décimal
= 1 * 23 + 1 * 22 + 0 * 21 + 1 * 20 on ne conserve que les coefficients
= 1 1 0 1 en binaire

Représentation des informations en binaire

Pour coder de l'information , que ce soient des nombres, (PI=3,141592...), du texte (ce cours), des schémas, des images, des sons, des relations ("Pierre est le pére de Jacques et le frêre de Marie"), les circuits électroniques d'un ordinateur ne peuvent utiliser que des mots en binaire.

Montrons d'abord comment il est possible de coder n'importe quel nombre entier naturel IN={0, 1, 2, 3, ... ,1 000, ..., 1 000 000, ...} en binaire.

Puis nous en ferons autant pour les lettres et les mots de la langue française. Enfin il faudra montrer que les images, les sons et les relations aussi peuvent se coder en binaire.

Passer du décimal au binaire

Il suffit de décomposer un nombre décimal en une somme de puissances de 2. On peut par exemple commencer par écrire la table des premières puissances de 2 :

$2^0 = 1$	$2^1 = 2$	$2^2 = 2x2 = 4$	$2^3 = 2x2x2 = 8$
$2^4 =$	$2^5 = 2x...x2 =$	$2^6 =$	$2^7 =$
$2^8 =$	$2^9 =$	$2^{10} =$	$2^{11} =$

Exercice 1 : Montrer que le nombre 256 est une puissance de 2.
Exercice 2 : Montrer que le nombre 131 est une somme de puissances de 2

Exercice 3 : Donner la représentation binaire des 10 premiers nombres entiers :

0 = $0x2^0$ -> 0; 1=$1x2^0$ -> 1 ; 2=$1x2^1$ -> 10

3 = 1x2 + 1x1 = $1x2^1 + 1x2^0$ -> 11

4 = 1x4+ 0x2 + 0x1 = $1x2^2 + 0x2^1 + 0x2^0$-> 100

5 = 1x4 + 0x2 + 1x1 =

6 =

7 =

8 =

9 =

10 =

Passer du binaire au décimal

Le codage binaire a un inconvénient majeur pour l'être humain, son manque de lisibilité...
Quelle est la représentation décimale du nombre dont la représentation binaire est : 1001 1110 ?

Réponse : $1x2^7+ 0x2^6 + 0x2^5 +1x2^4 + 1x2^3 + 1x2^2 + 1x2^1 + 0x2^0$

= 1x128 + 0x64 + 0x32 +1x16 + 1x8 + 1x4 + 1x2 + 0x1

= 158

Exercice 4 : Même question pour 1111 1111

Exercice 5 : Combien de chiffres binaires faut-il pour représenter le nombre décimal 10000 ?

Opérations usuelles en binaire

Les deux opérations binaires de base sont
- l'addition
- la multiplication

Table d'addition binaire

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 avec une *retenue* de 1

Table de multiplication binaire

0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1

Exercice 6 :

En utilisant la table d'addition binaire calculez en binaire les nombres suivants :

1001 + 10100
1111 + 1
1010 + 111
1111 1111 + 1111 1111

Exercice 7 : En utilisant la table de multiplication et d'addition binaires calculez en binaire les nombres suivants :

1001 x 1
1111 x 10
100 x 101
1111 x 1111

Opérations logiques

En logique binaire une variable logique (booléenne) est VRAI (TRUE = 1) ou FAUSSE (FALSE = 0). Une expression logique est constituée de plusieurs variables logiques combinées par des connecteurs (opérateurs) logiques :

Les opérateurs logiques élémentaires sont :
- NON [*NOT*]
- ET [*AND*]
- OU [*OR*]
- OU EXCLUSIF [*XOR*]

Table de vérité du NON

NON 0 = 1
NON 1 = 0

Table de vérité du OU

0 OU 0 = 0
0 OU 1 = 1
1 OU 0 = 1
1 OU 1 = 1

Table de vérité du ET

0 ET 0 = 0
0 ET 1 = 0
1 ET 0 = 0
1 ET 1 = 1

Table de vérité du OU EXCLUSIF [*XOR*]

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

Exercice 8 : Donner la valeur de vérité {VRAI ou FAUX} des assertions suivantes :
A : « 102 est un nombre pair »
B : « 11 est un multiple de 3 »
C : « 102 est un nombre pair » ET « 102 est divisible par 3 »
D : « 11 est multiple de 3 » ET « 102 est divisible par 3 »
E : « 108 est multiple de 9 » OU « 11 est divisible par 3 »

Les nombres entiers et les nombres réels décimaux

L'ensemble des entiers naturels : IN = {0,1,2..,100,101,...1000,...}
- IN est ordonné ; il a un plus petit élément 0 ;
- Il n'y a pas de plus grand élément : tout élément a un successeur

L'ensemble des entiers relatifs : Z = Z+ ∪ Z-

Z = {...,-1000,...,-100,..., -3,-2,-1, 0, 1,2..,100,101,...1000,...}
- Z est ordonné
- il n'y a ni plus grand ni plus petit élément : tout élément de Z a un prédécesseur et un successeur.

Opérations sur les entiers

Opérateurs unaires :
- (moins) : opposé
succ : successeur renvoie le nombre suivant dans l'ordre des entiers
pred : prédécesseur renvoie le nombre précédent
maxint : renvoie le plus grand entier représenté sur la machine

Opérateurs binaires :
+ - * DIV MOD /

Opérateurs de comparaison :
= < <=> >=

Axiomes :
associativité de + et *; distributivité de * sur +; relation d'ordre

Implémentation des entiers non signés

En raison des limitations d'espace mémoire, on ne peut représenter que des intervalles de nombres.
Deux difficultés à résoudre : choix des intervalles et représentation des nombres négatifs
Implantation courante sur 2 octets: l'intervalle sélectionné est [0..65535]
Il s'agit de générer une représentation de ce nombre en base 2 occupant au plus deux octets, soit 16 bits.

Exemple pour n = 547_{10}
On décompose 547 en puissances successives de 2 :

$547_{10} = 512 + 32 + 2 + 1 = 1*2^9 + 1*2^5 + 1*2^1 + 1*2^0$

et on ne code que les coefficients de la décomposition, la représentation binaire de n est

Bin(n) = 0000 0010 0010 0011

Le plus grand nombre entier non signé représentable est donc :
1111 1111 1111 1111

$= 1*2^{15} + 1*2^{14} + 1*2^{13} + 1*2^{11} + 1*2^{10} + 1*2^9 + 1*2^8$
$+ 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$
$= 1*2^{16} - 1$

La représentation décimale de ce nombre est donc
Dec(n) = 65 536 - 1 = $65\,535_{10}$

Implémentation des entiers signés

L'intervalle sélectionné est [-32768.. +32767]

Soit un nombre n de Z. Il s'agit de générer une représentation de ce nombre en base 2 occupant au plus deux octets, soit 16 bits. Si le nombre est positif et inférieur à 2^{15} (32 768) on le représente comme un entier non signé. Si le nombre est négatif et supérieur à -32768, le problème est de représenter le signe et de pouvoir passer d'un nombre à son opposé de façon simple.
Une méthode très répandue est la méthode du complément à 2. On travaille MODULO 2^{16}
On représente Bin(2^{16} + n) = 65 536 - 547 = 64 989
= 1111 1101 1101 1101
Une vérification de l'expression calculée consiste à effectuer une somme bit à bit de 547 et de -547
Si la représentation est correcte on doit trouver 0 :

```
  0000 0010 0010 0011
+ 1111 1101 1101 1101
  ___________________
Report □   0000 0000 0000 0000
```

Le report (ou *retenue*) □n'est bien sûr pas représenté.
Donc un nombre entier signé représenté sur 16 bits dont le bit de poids fort est à 1 doit être considéré comme un nombre NEGATIF : sa valeur est $-(2^{16} - Dec(n_2))$

Algorithme de conversion d'un nombre en binaire complément à 2 sur un octet :

Trouver l'opposé d'un nombre en complément à 2 :
Soit n = 0000 0101 = 5_{10}
Son opposé est -5 obtenu en inversant tous les 1 en 0 et tous les 0 en 1 et en ajoutant 1:
0000 0101 --> 1111 1010 + 1 = 1111 1011 = -5

Vérification :

```
              5 + (-5) = 0
       0000 0101
      +1111 1011
      ----------
    □  0000 0000  report □□ignoré
```

Exercice :

Trouver les représentations binaires en complément à deux sur deux octets des nombres suivants :

-1; -2; 31000; -31000

-33000 est-il représentable ?

Les nombres réels.

L'ensemble IR ne peut pas être représenté de façon compléte en machine.

On retrouve les mêmes difficultés pour représenter les réels que pour représenter les entiers : l'ordinateur n'a pas une mémoire infinie. On représente donc un sous-ensemble fini de IR. Tout traitement (opération, représentation) peut être entachée d'erreur. Un calcul mathématique permet d'estimer l'importance de cette erreur.

Opérations sur les réels :

Opération unaire : - (opposé)

Opérations binaires : + - * /

Comparaisons = < > <= >=

Fonctions réelles : cos sin log puiss sqrt Axiomes

Corps ordonné

x * (y + z) = x * y + x * z : distributivité

On retrouve certains axiomes des entiers plus ceux des rationnels (inverse)

plus quelques caractéristiques intéressantes (densité)

La notation scientifique normalisée

On appelle notation normalisée d'un réel celle où le premier chiffre significatif est placé immédiatement après la virgule (ou le point décimal).

Exemple : 1989 = 0.1989 E4

Pour stocker ce nombre en mémoire, il suffit de stocker l'exposant 4 et la partie décimale appelée mantisse 1989

Exemple : écrire PI en notation normalisée π = 3.1415926535... = 0.31415926535 E1

Représentation des nombres réels en binaire

Tout nombre réel peut être représenté dans une base quelconque :

$a = M * B^{e}$

avec B : base ; e : exposant ; M : mantisse (qui peut être une suite infinie de chiffres...)

En notation normalisée on a les conditions :

$(1/B) <= | M | < 1$ ou bien M=0

Les réels sont représentés en machine selon un standard défini par l' IEEE

Un réel est décomposé en

signe	+ -	**s**
exposant	caractéristique	**E**
partie décimale	mantisse	**F**

$x = (-1)^{\mathbf{s}} . 2^{\mathbf{E}-127} . 1, \mathbf{F}$

En général on représente un réel sur 4 octets (32 bits)

le bit 0 de poids fort (le plus à gauche) est le signe **s**, soit 0 (positif) ou 1 (négatif)

les bits 1 à 8 sont la caractéristique **E** qui est représentée par un entier binaire sur 8 bits décalé de la valeur 127

les bits 9 à 31 sont pour exprimer la mantisse **F** en binaire,

Exemple : 0.8 sera codé :

```
S=0 E=126   F
. ........  .......................
0 01111110  10011001100110011001100   codage binaire
. ........  .......................
0 1  8  9       31    n° de bit
```

Décomposition d'un nombre réel décimal en binaire

Soit le nombre 0.8 à convertir en binaire. On constate d'abord que son signe est positif, donc S=0. On cherche ensuite à le décomposer en une somme de puissances de 2.

$0.8 = 2^0 \times 0.8$

$0.8 \times 2 = 1.6$ donc $0.8 = 1.6 / 2 = 2^{-1} \times 1.6 = 2^{-1} \times 1 + 2^{-1} \times 0.6$

$0.6 \times 2 = 1.2$ donc $0.8 = 2^{-1} \times 1 + 2^{-2} \times 1.2$

$0.2 \times 2 = 0.4$ donc $0.8 = 2^{-1} \times 1 + 2^{-2} \times 1 + 2^{-3} \times 0.4$

$0.4 \times 2 = 0.8$ donc $0.8 = 2^{-1} \times 1 + 2^{-2} \times 1 + 2^{-3} \times 0 + 2^{-4} \times 0.8$

$0.8 \times 2 = 1.6$ donc ... on retrouve une expression déjà rencontrée qui va se répéter infiniment

$0.8 = (-1)^0 \times 1.\ 10011001100110011.... \times 2^{-1}$

Finalement on ne code que les chiffres binaires de la décomposition :

signe = 0 car 0.8 est positif

E = 126 car 126-127 = -1

F = 10011001100110011....

Exemple 2 : 0.75 en base 2 donnera

$0.75 \times 2 = 1.5$ donc $0.75 = 2^{-1} \times 1.5$

$0.5 \times 2 = 1.0$ donc $0.75 = 2^{-1} \times 1 + 2^{-2} \times 1.0$

$0.75 = (-1)° \times 2^{126-127} \times 1.10000000000000000000000$

Exercice : Montrer que le nombre décimal 0.1 n'a pas de représentation finie en binaire.

Réels en double précision

En augmentant le nombre d'octets attribués à la représentation des réels, on améliore la précision en consacrant plus de bits à la mantisse, mais on n'augmente pas l'intervalle des réels représentés car on conserve le même nombre de bits pour la caractéristique.

Les nombres rationnels

La représentation des réels a de gros défauts dés qu'il s'agit par exemple de comparer des nombres manifestement égaux mais dont on ignore si l'ordinateur les identifiera.

x = 0.000 020 et y = 0.000 019 999...;

u = 20 ; v = 19 999...

Comment est représenté z = (x/y) et r = (u/v) ? Sont-ils perçus comme égaux ?

Il peut être avantageux de définir directement un type de données NOMBRE RATIONNEL qui représentera de façon exacte tous les nombres équivalents à une fraction entiére...

Codage des informations non numériques

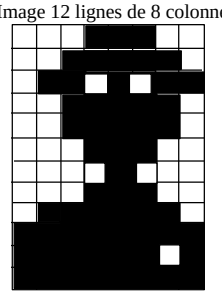
Textes

Les textes peuvent être représentés en binaire à condition d'associer à chaque lettre de l'alphabet une représentation numérique, par exemple son rang dans l'ordre alphabétique : A serait codé1 ; B, de rang 2 serait codé 10 en binaire; C codé 11, etc. Mais ce codage est insuffisant car il faut aussi pouvoir coder les signes de ponctuation . , ; ? ! , distinguer les minuscules des MAJUSCULES, les caractères accentués, etc.
La table ASCII (American Standard Code for Interexchange Information) propose un codage de 128 caractères différents sur 7 bits. Dans la table ASCII la lettre A est codé 65 B est codé 66 C est codé 67 ... Z est codé 90 a est codé 97 b est codé 98 ... 0 est codé 48 1 est codé 49 ... 9 est codé 57 ...
Les caractères accentués des langues européennes ont nécessité l'ajout d'un huitiéme bit de codage, d'où la table ASCII étendue avec 256 caractères. Mais ce n'était pas suffisant pour certaines langues comme le japonais ; un codage sur 16 bits est en cours de normalisation sous le nom d'UNICODE. Il supplantera dans l'avenir le code ASCII.

Les images

Le codage des images en noir et blanc ne présente pas de difficulté. Il suffit d'observer à la loupe une photographie de journal pour en comprendre le principe. Si à une image est superposée une grille très fine, chaque carré de la grille coloré en NOIR par l'image est codé 1, chaque carré BLANC est codé 0. Il suffit donc de coder toute l'image sous forme d'un tableau à deux dimensions de 0 et de 1.

Image 12 lignes de 8 colonnes



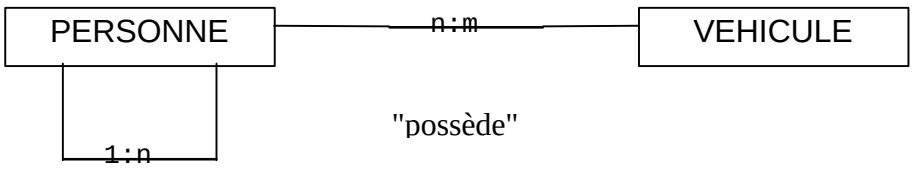
Recodage binaire

0001 1100
0011 1110
0110 1011
0011 1110
0011 1110
0001 1100
0000 1000
0001 1100
0111 1110
1111 1111
1111 1101
1111 1111

Le codage des sons est plus compliqué. Il faut d'abord transformer chaque son en un signal électrique continu (c'est le rôle du microphone), puis échantillonner ce signal électrique (discrétiser) et le numériser. On dispose alors d'un série de nombres qui représentent le signal sonore...

Les relations

Le cours de bases de données (seconde année GTE) introduit le modéle entité/association qui permet de représenter les informations "Marie est la mére de Pierre et de Françoise" ; "Pierre et Françoise ont acheté ensemble le véhicule Renault Clio 1234 ZA 75", "Marie a une Citroën ZX" ; "la Twingo 3987 TT 51 n'est à personne", etc., sous forme de tables relationnelles.



PERSONNE

Nom	*Mére*
Marie	?
Pierre	Marie
Françoise	Marie

POSSEDE

Propriétaire	*Véhicule*
Pierre	1234 ZA 75
Françoise	1234 ZA 75
Marie	1001 AR 34

VEHICULE

Immatriculation	*Marque*	*Modéle*
1234 ZA 75	Renault	Clio
1001 AR 34	Citroën	ZX
3987 TT 51	Renault	Twingo

Notion d'algorithme

Un algorithme est un procédé automatique qui transforme une information symbolique en une autre information symbolique. Seuls les problèmes qui sont susceptibles d'être résolus par un algorithme sont accessibles aux ordinateurs.

DONNEES	---- transformation ------>	RESULTAT
Entrée	----- algorithme --------->	Sortie

Ce qui caractérise l'exécution d'un algorithme, c'est la réalisation d'un nombre fini d'opérations élémentaires (instructions) ; chacune d'elles est réalisable en un temps fini. La quantité de données manipulées au cours du traitement est donc finie.

La notion d'opération élémentaire dépend du degré de raffinement adopté pour la description du procédé. Ainsi, chaque algorithme peut être considéré comme une opération élémentaire dans un procédé plus important.

Exemple d'algorithme : Factorisation de ax^2+bx+c quand $a \neq 0$.

Algorithme A :

soient x_1 et x_2 les zéros de ax^2+bx+c ;

alors $ax^2+bx+c = a\,(x-x_1)\,(x-x_2)$.

Algorithme B

soit $\Delta = b^2-4ac$;

si $\Delta = 0$ alors soient x_1 et x_2 égaux à $-b/2a$

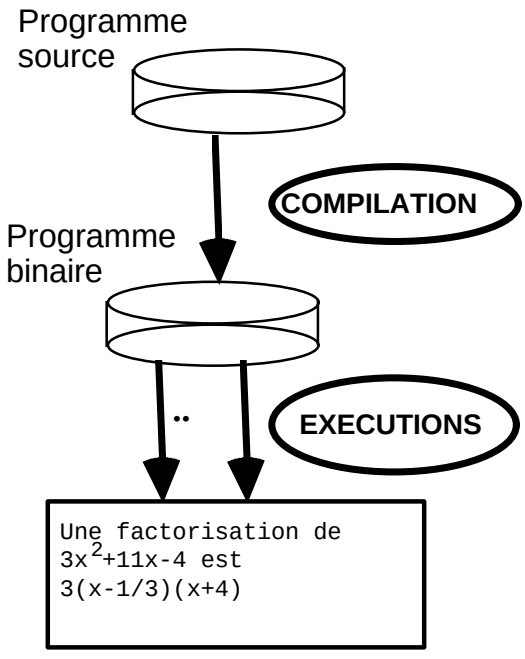
sinon si $\Delta > 0$ alors soient $x_1=(-b+\sqrt{\Delta})/2a$ et $x_2= (-b-\sqrt{\Delta})/2a$

sinon soient $x_1=(-b+i\sqrt{(-\Delta)})/2a$ et $x_2= (-b-i\sqrt{(-\Delta)})/2a$;

alors $ax^2+bx+c = a\,(x-x_1)\,(x-x_2)$.

Traduction de l'algorithme dans un langage de programmation

Avant de faire traiter la factorisation de ax^2+bx+c quand $a \neq 0$ par un ordinateur, il faut traduire cet algorithme dans le langage binaire susceptible d'être exécuté par la machine. Cette transformation est le travail du programmeur. Il dispose pour cela d'un langage de programmation dit de haut niveau, c'est-à-dire qu'un être humain peut apprendre et manipuler sans trop de difficultés, et de programmes spécialisés, appelés compilateurs, qui font la conversion d'un fichier écrit dans le langage de haut niveau en code binaire exécutable par la machine cible. Une fois compilé, le programme (s'il est correct) peut être exécuté de multiples fois. Les résultats de son exécution (les sorties) dépendent des paramètres fournis en entrée.



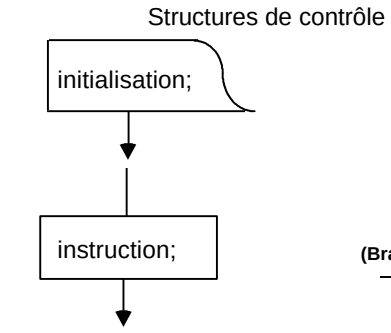
Avant de passer à la phase de programmation il est donc nécessaire de définir très précisément le cahier des charges du programme, c'est-à-dire dans quelles conditions initiales il devra fonctionner, comment il devra procéder (algorithme) et sous quelle forme seront présentés les résultats.

Types de données et structures de contrôle

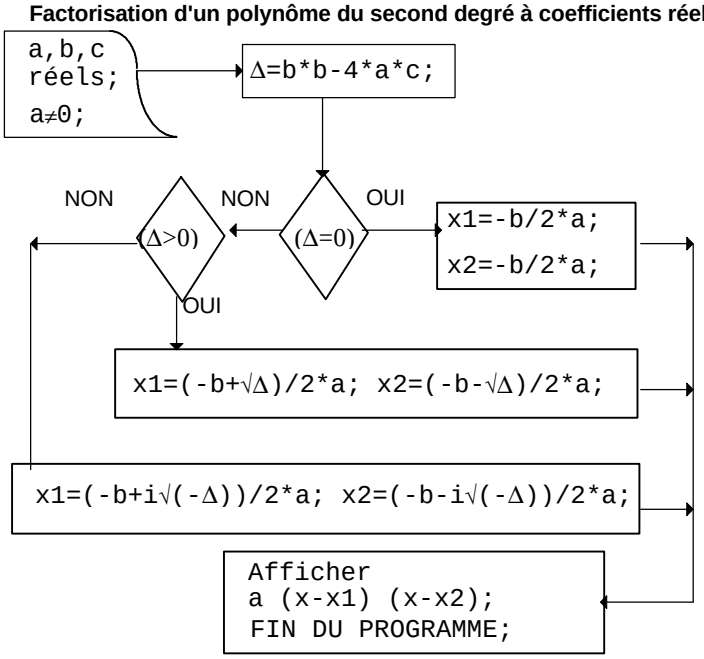
Dans l'exemple de la factorisation ci-dessus, les entités manipulées sont des nombres (complexes et réels), des coefficient constants (a, b, c), une "inconnue" (x), des variables x_1 et x_2, le symbole Δ du discriminant et les opérations élémentaires sur l'ensemble des nombres réels (<, >, =, +, -, *, /). On dira que les **types de données** sont des **constantes** et des **variables** de **type** réel. L'algorithme emploie aussi des **structures de contrôle** conditionnelles : **Si** (*condition*) **alors** *instruction* **sinon** *instruction*.. Un instruction est soit une affectation — $\Delta = b^2-4ac$; —, soit un test— si $\Delta = 0$ alors — .Le langage de programmation devra fournir des équivalents de toutes ces entités. Enfin le langage doit permettre de créer un programme qui reçoive des paramètres —saisie de la valeur des coefficients— et retourne des résultats à l'utilisateur —affichage—.

Un langage de programmation graphique

Traduisons d'abord l'algorithme de factorisation dans un langage graphique élémentaire, bien adapté à l'expression des algorithmes peu complexes.



Factorisation d'un polynôme du second degré à coefficients réels



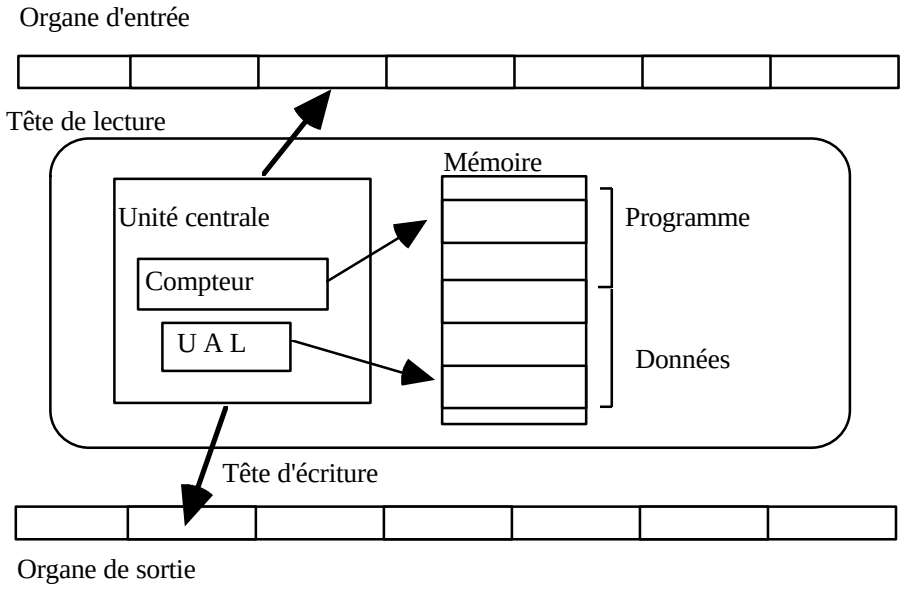
Chaque boîte peut contenir une ou plusieurs instructions exécutées séquentiellement, c'est-à-dire successivement dans l'ordre de lecture de haut en bas. Un test est une expression de type *booléen*, à savoir prenant la valeur VRAI ou la valeur FAUX lors de son évaluation. Enfin les branchements sont parcourus dans le sens des fléches.

Il suffit d'exécuter chaque instruction à la lettre, en suivant les fléches désignées à la suite de chaque test pour résoudre le problème de factorisation. On constate que cette représentation fait l'économie d'un test par rapport à la résolution mathématique.

Cependant peu nombreux sont les programmes capables d'interpréter directement un langage graphique, aussi faut-il exprimer le programme dans un langage littéral.

Modéle d'ordinateur abstrait et langage de programmation

Passons donc à la traduction de l'algorithme dans un langage de programmation élémentaire, le langage de la machine RAM (Random Acces Machine : Machine à Accès Direct, voir le cours d'algorithmique).
Les algorithmes sont écrits pour une machine abstraite qui a les caractéristiques d'une machine de Von Neumann. Elle comprend :
- des organes d'entrée et de sortie (interface avec l'utilisateur) ;
- une mémoire unique pour le programme et les données ;
- une unité centrale comportant notamment un compteur d'instruction, un processeur arithmétique et logique, des registres et des têtes de lecture et d'écriture et un moyen d'accès à la mémoire ;
- un jeu d'instructions exécutables par l'unité centrale.



La mémoire est une suite de cases dont les indices sont appelés *adresses*. Une partie de la mémoire contient le *programme*, traduction de l'algorithme au moyen des instructions de la machine. Le compteur d'instruction contient l'adresse de la prochaine instruction à exécuter. L'autre partie de la mémoire contient les données.

Caractéristiques :
- l'unité centrale *accéde directement* (en temps constant) à une case mémoire à partir de son *adresse* [*random acces*].
- La *mémoire est infinie* (mais le programme est fini).
- Chaque case mémoire contient une *donnée élémentaire* de taille arbitraire. On peut ainsi y mémoriser des entiers arbitrairement grands.
- Chaque instruction s'exécute en *temps constant*.

Variables et types

Une case mémoire de la machine abstraite est aussi appelée une *variable*. Dans les programmes on désigne une variable par un *identificateur* littéral plutôt que par son adresse, et, par abus de langage, on dit "la variable *x* " plutôt que "la variable d'identificateur *x* ". Le *type* d'une variable définit l'ensemble des valeurs qu'elle peut prendre —du point de vue implantation en machine, le type décide aussi de la taille de l'espace mémoire occupé par la donnée.

Types simples et types combinés

Les types simples sont
- l'entier {....,-1 000 000,... -1 000,... -3,-2,-1, 0, 1, 2, 3, ..10, ..,100, .., 1 000, ...}
- le booléen {VRAI, FAUX} [*boolean* {*TRUE, FALSE*}]
- le caractère {'0','1','2',..,'9',..,'A','B',..,'Z',..,'a','b',...}
- l'énuméré (Ex.: [violet,bleu,vert,jaune,rouge])
- le 'réel' flottant (Ex.: valeur approchée de PI 0.3141592 E+1)

A partir des types simples des opérateurs permettent de construire des types plus complexes :
- couples d'entiers (rationnels)
- couples de flottants (nombres complexes)
- chaînes de caractères (Ex.: "CECI EST UNE CHAINE DE CARACTERES")
- tableaux (Ex.: les 8 premières puissances de 2 en base 10 : [1, 2, 4, 8, 16, 32, 64, 128])
- ensembles (Ex.: ensemble de couples d'entiers :{ (0,1), (1,2), (2,4), (3,8), (4,16), (5,32), (6,64), (7,128)}
- structures (Ex.: fiche de répertoire : (Nom, Prénom, Date de naissance, Adresse, Téléphone))

Sur chaque type est défini un domaine et un ensemble d'opérations :
- Addition, multiplication, successeur sur les entiers naturels
- Addition, soustraction, multiplication, successeur, prédécesseur, modulo sur les entiers relatifs
- Addition, soustraction, multiplication, division, exponentiation, logarithme sur les réels
- Opérations logiques ET, OU, NON sur les booléens
- Union, intersection, complémentation, produit cartésien sur les ensembles
- Concaténation sur les chaînes de caractères,

etc.

Opérations de bases

Les opérations de base sur tous les types de variables sont les opérations d'*entrée-sortie* et l'*affectation*.. Soient *x* et *y* des identificateurs de variables.

lire(x) signifie : copier la valeur qui est en face de la tête de lecture (sur l'organe d'entrée) dans la case-mémoire identifiée par *x* ; puis placer la tête de lecture sur la donnée suivante.

écrire(x) signifie : copier la valeur contenue dans la case mémoire identifiée par *x* sur l'organe de sortie en face de la tête d'écriture ; puis avancer cette tête.

x = expression se lit "*x* reçoit *expression*" et signifie : évaluer l'expression et placer la valeur obtenue dans la case-mémoire identifiée par *x*.. Si *y* apparaît dans l'expression, la valeur contenue dans la case-mémoire correspondant à *y* est utilisée pour l'évaluation :
Ex : *x = x + y* : Le contenu de la case mémoire désignée par *x* est remplacé par la somme des valeurs contenues dans les cases mémoires désignées par *x* et par *y*. Le contenu original de *x* est perdu, celui de *y* est conservé. ([2])

Les structures de contrôle du langage
- composition séquentielle
  { instruction1; instruction2; ...; instructionN;}
- composition conditionnelle :
  {SI (condition) ALORS (instruction1) SINON (instruction2)}
- composition itérative
  {POUR (énumération) FAIRE (instruction)}
  {TANT QUE (test) FAIRE (instruction)}
- des appels de fonctions et sous-programmes .

[2]Pour préparer le lecteur aux conventions du langage C, nous conviendrons de noter l'affectation par un signe '=' et l'égalité par '=='.

De façon plus précise la syntaxe du langage est définie par la grammaire formelle suivante (simplifiée du langage C) :

<programme> ::=	type du résultat *identificateur* (type *paramètre*); { *suite d'instructions séparées par des points-virgules* }
<instruction> ::=	<instruction élémentaire> | { *suite d'instructions séparées par des points-virgules* } | **si** (*test*) <instruction> **sinon** <instruction> | **pour** (*i =d à f*) **faire** <instruction> | **tant que** (*test*) **faire** <instruction>
<instruction élémentaire> ::=	*variable =expression* | **lire***(identificateur)* | **écrire***(expression)* | **retour**(*expression*)**;**

L'instruction **retour**(*expression*)**;** a pour effet d'arrêter l'exécution de l'algorithme et de produire la valeur de l'expression. Une expression est bien parenthésée au sens de l'algèbre et prend une valeur du type des éléments qui la composent.

Avec ce langage, traduisons l'algorithme de factorisation. Nous obtenons une fonction qui retourne un couple de polynômes (éventuellement complexes) de degré 1.

Fonction factorisation d'un polynôme réel de degré 2

```
couple de polynômes de degré 1 factorisation ( polynôme réel de degré 2 ax²+bx+c);
{
    réel D;
    D =b²-4ac
    si (D==0)
        retour((x+b/2a), (x+b/2a);
    sinon  si (D>0)
            retour((x-(-b+ÖD)/2a), (x-(-b- D)/2a)  );
        sinon
            retour((x-(-b+iÖ (-D))/2a), (x-(-b-iÖ (-D))/2a));
}
```

Traduit dans le langage de la machine RAM, l'algorithme de factorisation reste encore très proche de ses origines mathématiques. L'ultime transformation va le traduire dans un langage effectif, le langage C, pour obtenir un programme exécutable par une machine réelle.

Conclusion

Cet exposé de la factorisation d'un polynôme nous a permis de passer par les trois étapes de création d'un programme numérique :

définition du problème,

rédaction de l'algorithme en termes mathématiques,

traduction dans un langage de programmation (la traduction en langage C sera abordée au chapitre suivant).

Il faut insister ici sur l'extréme rigueur de la syntaxe des langages de programmation. Toutes les constantes, variables et fonctions utilisées doivent être typées et définies avant appel. Les instructions respectent une grammaire précise, qui caractérise le langage ; les fonctions disponibles (plusieurs centaines en C) fournissent une grande variété d'outils dont la maîtrise ne peut s'acquérir que peu à peu... Cet apprentissage ne doit pas être confondu avec une formation à l'art de la programmation qui peut débuter avec des exercices plus simples et moins rebutants.

Le langage C.

Le langage C est un langage d'ingénieur destiné à la création d'applications informatiques. Beaucoup d'ouvrages lui ont été consacrés. Le plus important est dû aux créateurs du langage eux-même, Denis Ritchie et Brian Kerninghan. On en trouvera la référence dans la bibliographie. Un programme en C est constitué d'un (ou plusieurs) fichiers sources organisés d'une façon conventionnelle. Voici une traduction en C de l'algorithme de factorisation. Les cadres (qui ne sont pas nécessaires dans un programme, mais permettent ici de fixer les idées) délimitent cinq parties fonctionnellement interdépendantes du fichier source.

```
/* entête : fichiers inclus */
#include <stdio.h>
#include <math.h> /* etc. */
```
(1)

```
/* déclarations de constantes
et de variables globales */
#define FALSE 0
#define TRUE 1
float a, b, c;
```
(2)

```
/* prototypes de fonctions */
void factorisation(float a, float b, float c);
```
(3)

```
void main (void) /* programme principal */
{
    printf("Factorisation d'un polynôme réel de
degré 2\n");
    printf("Entrez trois nombres réels a, b, c
(a°0)\n");
    scanf("%f %f %f", &a, &b, &c);
    if (a==0) {
        printf("Erreur de saisie\n");
        exit(0);
    }
    factorisation(a, b, c);
}  /* fin du programme */
```
(4)

```
/* définition des fonctions */
 void factorisation(float a, float b, float c)
 /* on assume que a est non nul */
 {
    float delta = b*b - 4*a*c;
    /* delta est le discriminant */
    /* c'est une variable locale */
    printf("La factorisation donne \n");
    if (delta==0)
        printf(" (%f)(x-(%f))(x-(%f))\n",
                a,-b/(2*a),-b/(2*a));
    else if (delta>0)
       printf(" (%f)(x-(%f))(x-(%f))\n",
       a,(-b-sqrt(delta))/2*a, (-b+sqrt(delta))/2*a);
    else
       printf(" (%f)(x+(%f)+(%f)i)(x+(%f)-(%f)i)\n",
       a, b/(2*a), sqrt(-delta)/2*a, b/(2*a),
 sqrt(-delta)/2*a);
 }
```
(5)

Le bloc (1) est celui des fichiers inclus. Sa première ligne, `#include <stdio.h>`, invoque la bibliothèque des fonctions d'entrée-sortie (saisie au clavier `scanf()` et affichage à l'écran

`printf()`). La deuxiéme ligne, `#include <math.h>`, fait appel aux fonctions mathématiques (`sqrt()` : racine carrée).

Vient ensuite —bloc (2)— la définition des constantes et des variables *globales*, c'est-à-dire vues depuis tous les points du programme :

```
#define FALSE 0
#define TRUE 1
float a, b, c;
```

Puis on trouve le *prototype* de la fonction *factorisation()* :

```
void factorisation (float a, float b, float c);
```

Celle-ci prend trois paramètres —a, b, c : les coefficients du polynôme à factoriser— de type `float`; mais comme elle ne retourne aucune valeur, elle est typée `void`.

Enfin c'est le bloc (4) de la fonction `main()`. C'est le point d'entrée du programme. Tout programme en Langage C a une fonction main() et une seule. Celle-ci affiche deux lignes de message et lit ensuite le clavier —`scanf("%f %f %f", &a, &b, &c);`— jusqu'à l'entrée de trois nombres 'flottants'. Après avoir testé la condition (a 0) la fonction `factorisation()` est appelée et le programme se termine.

Le dernier bloc (5) est le code de la fonction `factorisation()`. Le lecteur reconnaîtra la définition du discriminant et l'expression des différentes factorisations selon la valeur de . Nous n'entrerons pas maintenant dans le détail de la syntaxe des fonctions `printf()` et `scanf()`, qui sont parmi les plus compliquées du langage C. Je renvoie le lecteur aux ouvrages cités en référence et au support du cours de langage C.

Les étapes suivantes consistent à *compiler* ce *programme source*, puis à *lier* le fichier objet obtenu après compilation avec les bibliothèques standard et mathématique, ce qui produit un *programme exécutable*. En cas d'erreur, ou pour modifier ce programme, il faut reprendre toute la séquence en rééditant le fichier source...
C est étroitement associé à UNIX Le Système d'Exploitation (Operating System) UNIX développé aux Laboratoires Bell (ATT Corporation - USA) par B.W. Kernigham et D.M. Ritchie dans les années 70, a été écrit en C, développé pour l'occasion.
Unix est multi-tâches et multi-utilisateurs, sur mini et stations de travail. Sa diffusion a assuré le succés de C chez les universitaires et les ingénieurs.
Unix est aujourd'hui fortement concurrent d'OS2 sur micros puissants...

Caractéristiques succintes du langage C

Le langage C est un langage des années 70, donc 'moderne' par rapport à FORTRAN ou BASIC, *procédural* (description linéaire des états de la mémoire, comme Fortran, Basic, Lisp, Pascal..., contrairement aux langages déclaratifs comme SQL ou Prolog (ensemble de faits et de règles + moteur d'inférence).

C est structuré en *blocs* fonctionnels imbriqués. C fait appel à des variables *locales* à chaque bloc ou à des variables *globales* à plusieurs blocs.

C supporte l'appel de fonctions et c'est un langage *typé : les variables et les fonctions* utilisées doivent être déclarées et leur types et vérifié à la compilation. C est un langage *compilé* : le code source est transformé en code objet et lié pour produire un *exécutable.*

Comparé à PASCAL, C est plus CONCIS mais plus obscur. C est mieux standardisé, mais les compilateurs C sont plus libéraux que les compilateurs Pascal pour la vérification des types de données.

Pour s'assurer du portage d'un programme en C sur d'autres compilateurs, il est fortement conseillé de respecter les spécifications ANSI et de tenir compte des avertissements (warnings) à la compilation.

C est le langages des programmeurs système... mais sa disponibilité sur tous les systèmes informatiques en fait un langage indispensable (avec Fortran pour les applications scientifiques).

C n'est pas un langage à objets comme C++ ou Java.

Evolutions

Le futur de C est lié à la programmation parallèle, au développement des réseaux et à la Programmation Orientée Objets (C++, Java). L'apprentissage de C est un bon investissement pour l'ingénieur logiciel ou le chercheur amené à utiliser des stations de travail, à condition de programmer souvent.

Structures en blocs

Un bloc est une séquence d'une ou plusieurs instructions commençant par une accolade ouvrante { terminée par une accolade fermante }.

Exemple de structure en blocs pour un programme de jeu d'échecs

```
/* Programme Jeu d'échecs */

Initialiser_Variables
Initialiser_Affichage
TANT_QUE (La_Partie_Continue)
{
   SI (C_est_mon_Tour)
   {
    Déterminer_mouvement_Suivant ();
    Mettre_à_jour_Affichage ();
    Mettre_à_jour_Variables ();
   }
   SINON
   {
    Attendre_Déplacement_Adverse();
    Vérifier_Validité();
    Mettre_à_jour_Affichage();
    Mettre_à_jour_Variables();
   }
  }
```

Chaque Fonction() peut à son tour être décomposée en blocs :

```
/*             Fonction             */
Déterminer_mouvement_Suivant()
{
 Chercher_Tous_les_coups_Autorisés();
 POUR (Tous_ces_Coups)
  {
   Evaluer_la_Position();
   SI (Meilleure_Position_Jusque_là)
     {

Mettre_à_Jour_le_Mouvement_Sélectionn
é();
     }
  }
 RENVOI (Mouvement_Sélectionné);
}
```

Variables locales et variables globales

Dans chaque bloc il est possible de définir des **variables locales** - dont l'accès n'est licite qu'à l'intérieur du bloc considéré- dont la valeur est indéterminée sauf affectation explicite,
- dont l'emplacement mémoire est libéré à la sortie du bloc.
Les **variables globales** à plusieurs blocs sont connues
- dans le bloc englobant où elles sont définies,
- dans les blocs internes au bloc englobant, sauf en cas de masquage par une redéfinition sous le même nom de variable.
Dans l'exemple suivant on vérifie que la variable globale **i** vaut 1 en entrée et en sortie du premier bloc et n'est pas affectée par la variable **i** interne au deuxiéme bloc.

```
/* Structure en Blocs -- EXO1.C */
#include <stdio.h>
main() /* Programme principal */
{   /* Début du premier bloc */
int i, n;    /* Définition */
i=1;      /* Affectations */
n=5;
printf("Entrée du 1er Bloc\n");
printf("Variable i=%d\n",i);
printf("Nombre d'itérations %d\n",n);

{        /* second bloc */
  int i;     /*i Redéfini */
   /* Oubli de l'affectation */
  printf("Entrée du 2éme Bloc: i
redéfini et non affecté=%d\n",i);
  for (i=0; i<n; ++i)
   printf(" %d dans boucle\n",i);
  printf("Sortie du 2éme Bloc:
i=%d\n",i);
  }  /* Fin du second bloc */
printf("Sortie du 1er Bloc:
i=%d\n",i);
}    /* Fin du premier bloc */
```

L'exécution de ce programme produit :

```
Entrée du 1er Bloc
Variable i=1
Nombre d'itérations 5
Entrée du 2éme Bloc: i redéfini et
non affecté=80
 0 dans boucle
 1 dans boucle
 2 dans boucle
 3 dans boucle
 4 dans boucle
Sortie du 2éme Bloc: i=5
Sortie du 1er Bloc: i=1
```

Constantes et variables

Le langage C utilise les **constantes** numériques et les constantes caractères
- entiers : `0, 1, 2, -1, -2`, etc.
- flottants : `0.0, 0.3141592E1,`
- caractères : `'a', 'b', 'c',..., 'A', 'B', 'C',` etc.,
- constantes chaînes de caractères : `"CECI est une chaîne de caractères"`.

Les **variables** sont des adresses de la mémoire désignées par des **identificateurs** littéraux commençant par une lettre (exemple : `i, j, x, y, entier1, entier_2, bilan_energetique`)

Mots réservés

Les mots réservés ne peuvent pas servir de noms de variables.

`auto extern short break float sizeof case for static char`
`goto struct continue if switch default int typedef do long`
`union double register unsigned else return while`

Mots réservés supplémentaires pour la norme ANSI

`const signed volatile enum void`

Mots réservés supplémentaires pour le compilateur Turbo C

`asm huge pascal cdecl interrupt far near`

Types de données

Il faut **déclarer** le **type** de toutes les variables et de toutes les fonctions, qui indique à la fois l'intervalle de définition et les opérations licites

Types simples

Type	Signification	Taille (bits)	Valeurs limites
`int`	`entier`	16	`-32768 à +32768`
`short`	`entier`	16	`-32768 à +32768`
`long`	`entier`	32	`-2 147 483 648 à +2 147 483 648`
`char`	`caractère`	8	`-128..+127`
`float`	`réel`		`+-10 E-37 à +-10 E+38`
`double`	`réel`		`+-10 E-307 à +-10 E+308`

Une variable entiére peut être déclarée '**unsigned**'

unsigned int 16 0 .. 65535

Le type BOOLEAN est simulé en donnant la valeur **0** (FAUX) ou la valeur **1** (VRAI) à une variable entiére.

Les constantes de type caractère ont une valeur entiére dans la table ASCII

char c1 = 'A',
c2 = '**x**41'; /* représentation hexadécimale */

Caractères spéciaux

caractères	nom	symbole	code	code hexa	décimal
`\n`	`newline`	`LF`	`0A`	`10`	
`\t`	`tabulation`	`HT`	`09`	`9`	
`\b`	`backspace`	`BS`	`08`	`8`	
`\r`	`return`		`CR`	`0D`	`13`
`\f`	`form feed`	`FF`	`0C`	`12`	
`\a`	`bell`	`BEL`	`07`	`7`	
`\\`	`backslash`	`5C`	`92`		
`\'`	`single quote`	`27`	`39`		
`\"`	`double quote`	`22`	`34`		

Instruction d'affectation

L'assignation est une instruction qui **affecte** une valeur à une variable. Cette valeur peut provenir de l'évaluation d'une constante, d'une expression ou d'une fonction.
Le symbole = désigne l'affectation (assignation).

```
int   i, j, k;        /* déclaration */
  i = j = 5;    /* assignation de 5 à j et de j à i */
  k = 7;    /* assignation de 7 à k */
```

Transtypage (cast)

C permet des assignations entre variables de types différents.Une variable déclarée **char**, occupant un octet de mémoire, peut être **transtypée en int**, occupant deux octets de mémoire.

Régles de transtypage (casting)

char --> int le **char** se retrouve dans l'octet le moins significatif ; si le caractère a été déclaré unsigned char, il n'y a pas d'expansion du signe, sinon on retrouve le bit de poids fort répété 8 fois dans l'octet le plus significatif.

int --> char perte de l'octet le plus significatif

int --> long expansion du bit de signe

long --> int résultat tronqué (perte des deux octets les plus significatifs)

unsigned --> long les deux octets les plus significatifs sont mis à 0

int --> float exemple : **15 --> 15.0**

float --> int perte de la partie décimale : **2.5 --> 2** Si la partie entiére du réel est supérieure à 32767 le résultat sera aberrant.

float --> double pas de difficulté

double --> float perte de précision

Quand des expressions mélangent les types, le transtypage est automatique.

Opérateurs

C emploie plusieurs opérateurs : arithmétiques, de comparaison, logiques... La priorité des opérateurs entre eux permet d'évaluer d'une expression.

Opérateurs arithmétiques

Par ordre de priorité décroissante

Symbole	Signification		
*	/	multiplication	division (entiére et réelle)
+	-	addition	soustraction
%		reste de la division entiére (modulo)	

Opérateurs de comparaison

Si une expression est FAUSSE elle renvoie une valeur nulle , si elle est VRAIE elle renvoie une valeur non nulle.

Les opérateurs **&&** et || permettent de combiner des expressions logiques

Par ordre de priorité décroissante :

Symbole				Signification
!				NON (inverse une condition)
>	>=	<	<=	sup / sup ou egal / inf / inf ou egal
= =	!=			égal / différent
&&	||			ET logique / OU logique

Table de vérité de l'opérateur ET AND &&

FAUX	&&	FAUX	==	FAUX
FAUX	&&	VRAI	==	FAUX
VRAI	&&	FAUX	==	FAUX
VRAI	&&	VRAI	==	VRAI

Table de vérité de l'opérateur OU : OR : ||

FAUX	||	FAUX	==	FAUX
FAUX	||	VRAI	==	VRAI
VRAI	||	FAUX	==	VRAI
VRAI	||	VRAI	==	VRAI

Incrémentation et décrémentation

L'instruction *i = i* +1; remplace la valeur de i par i+1 ; c'est une *incrémentation* qui peut aussi s'écrire i++; ou ++i;

De même *i = i* **- 1;** peut s'écrire **i - -;** ou **- - i;**

Remarque: avec l'instruction **i++,** i est affecté puis incrémenté, avec **++i,** i est incrémenté puis affecté, avec l'instruction **i--,** i est affecté puis décrémenté, avec **- -i,** i est décrémenté puis affecté et enfin avec **i- -** , i est affecté puis décrémenté.

Si (e1) et (e2) sont des expressions et "op" une opération prise parmi la liste **+ - * / % << >> & | ^** alors
(e1) = (e1) op (e2); peut s'écrire **(e1) op = (e2);**
On économise une évaluation de e1.

Attention aux parenthèses : **x *= y + 1;**
est équivalent à **x = x * (y+1);** et non à **x = x * y + 1;**
Autrement dit si x==3, y==4 le résultat de cette instruction remplacera x par 15 et non pas par 13 !

Structures conditionnelles

Pour les structures conditionnelles, la condition évaluée doit être de type entier (**short, int, long**) ou **char**. Toute fonction qui renvoie une valeur d'un de ces types peut être testée dans la condition.

if / else

```
if (condition)  /* commentaire de la condition si */
    instruction;

if (condition)
 {  /* début de bloc*/
    instruction1;
    instruction2;
 }  /* fin de bloc */

if (condition)
    instruction_si;
else   /* sinon */
    instruction_sinon;
```

Conditions imbriquées

```
if (condition1)
    inst1;
else if (condition2)
          inst2;
     else if (condition3)
               inst3;
          else      /* chaque else se rapporte au if le plus proche sauf si on utilise des accolades comme dans l'exemple suivant */
               inst4;
```

Regroupement d'instructions

```
if (cond1)  /* premier if */
{
   if (cond2)
      inst1;
   else if (cond3)
         inst2;
}
else  /* sinon se rapportant au premier if */
   inst3;
```

Affectation conditionnelle

if (i>j**)** z = a**; else** z = b**;** est équivalent à z = (i>j) **?** a **:** b**;**

Sélection (switch)

L'instruction switch est une sorte d'aiguillage. Elle permet de remplacer plusieurs instructions imbriquées. La variable de contrôle est comparée à la valeur des constantes de chaque cas (**case**). Si la comparaison réussit, l'instruction du case est exécutée jusqu'à la première instruction break rencontrée.

```
switch (variable_controle)
{
 case valeur1 : instruction1;
    break;          /* sortie du case */
 case valeur2 : instruction2;
    break;
 case valeur3 :           /* plusieurs */
 case valeur4 :           /* étiquettes */
 case valeur5 : instruction3;  /* pour la même instruction */
    break;
 default : instruction4;     /* cas par défaut */
    break;       /* facultatif mais recommandé */
}
```

variable_controle doit être de type entier (**int, short, char, long**).
break fait sortir du sélecteur. En l'absence de **break**, l'instruction suivante est exécutée ; on peut ainsi tester plusieurs cas différents et leur attribuer la même instruction.

Boucles et sauts

Les boucles consistent à répéter plusieurs fois la même séquence d'instructions. La sortie de boucle est réalisée en testant une condition (de type booléen VRAI ou FAUX).

While, Tant que .

```
while (condition)
    instruction;

while (condition)
{
    instruction1;
    instruction2;
}
```

La condition est évaluée *avant* d'entrer dans la boucle. La boucle est répétée tant que la condition est VRAIE.

For, Pour.

```
for (initialisation; condition d'arrêt; incrémentation)
    instruction;
for (initialisation; condition d'arrêt; incrémentation)
{
    instruction1;
    instruction2;
    ...
}
```

La condition est évaluée *avant* d'entrer dans la boucle. L'incrémentation de la variable de contrôle est faite à la fin de chaque tour de boucle.

Exemple :

```
int i;
for (i=1; i<10; i++)
    printf("%d ",i);   /* ce programme affiche 1 2 3 4 5 6 7 8 9 */
```

Do ... While, Faire ... tant que.

```
do {
    instruction1;
    instruction2;
} while (condition d'arrêt);
```

La condition est évaluée *après* le passage dans la boucle.

Exit

Un programme peut être interrompu par l'instruction **exit(**code de retour**);** La valeur du code de retour est un entier qui peut être testée par le programme appelant.

Tableaux

Un tableau est une suite de cellules consécutives en mémoire pouvant contenir des données de type identique. La *taille du tableau* et le *type* des données doivent être déclarés en même temps que le tableau. Le nombre de dimensions n'est pas limité. L'indice (adresse relative de chaque cellule par rapport au début du tableau) doit être une expression entiére ; *la première cellule a l'indice 0*.

Tableaux à une dimension

Exemples :

int t**[**5**];**

Nom du tableau est *t* ; les cellules sont de type entier (**int**) ; la taille *5* ; indexée de 0 à 4

long l**[**2**];**

Nom du tableau est *l*; les cellules sont de type entier (**long**) ; la taille *2*; indexée de 0 à 1

char hexa**[**17**];**

Nom du tableau est *hexa* ; cellules de type **char** ; taille *17*

Un tableau peut être affecté en même temps qu'il est déclaré :

int tabint**[**5**]** = **{**0, 10, 100,**};** /* les autres cellules à 0 */

char binaire**[**2**]** = **{**'0', '1'**};**

Affectation

Pour initialiser un tableau, il faut désigner ses différentes cellules :

```
t[0] = tabint[4] = 10000;  /* affectation en cascade */
hexa[0] = '0';
for (i=0; i<5; i++)
  tabint[i] = i*i;
```

Attention : les dépassements des bornes d'indice des tableaux ne sont pas détectés à la compilation. C'est donc au programmeur d'ajouter les instructions de contrôle des bornes avant d'affecter une variable à un tableau.

Chaînes de caractères

Les chaînes sont des tableaux de caractères terminés par le caractère NULL ('**0**').

*Il faut déclarer leur **taille,** c'est-à-dire le nombre de caractères **plus un** pour stocker la fin de chaîne.*

char decimal**[**11**]** /* déclaration : 11 = 10 caractères + '\0' */

= "0123456789"; /* cette affectation : est équivalente à */

char decimal**[**11**]** = {'0','1','2','3','4','5','6','7','8','9','\0'};

char alpha[6] = "ABCDE" 5 caractères; composée de 6 octets identiques au code ASCII de chaque caractère sot 65 66 67 68 69 **00** en décimal, et 0x41 0x42 0x43 0x44 0x45 **0x00** en hexadécimal.

Tableaux à plusieurs dimensions

Deux dimensions (matrice)

int a**[**3**][**3**]** = {{1,2,-1},{-2,3,4},{5,0,6}};

a[0] désigne la 1ére ligne : {1,2,0}.

a[0][2] désigne le 3éme élément de la 1ére ligne : 1.

a[2][1] désigne le 2éme élément de la 3éme ligne : 0.

Trois dimensions

int parallelepipede[3][2][4]; 3 couches de 2 lignes de 4 colonnes chacune.

parallelepipede[0][0][0] désigne la première colonne de la première ligne de la première couche.

parallelepipede[2][1][3] désigne la dernière colonne de la dernière ligne de la dernière couche...

Pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable.

Selon le type de donnée contenu à l'adresse en question, on aura un **pointeur d'entier, de float, double, char**, ou de tout autre type. En accédant à cette adresse, on peut accéder indirectement à la variable et donc la modifier.

Déclaration de pointeur

Pour déclarer un variable pointeur on utilise l'opérateur ***** placé après le type du pointeur**.**
Un pointeur sur entier est déclaré par **int ***p**;**
Un pointeur sur réel **float ***p**;**
Un pointeur sur caractère **char ***p**;**
Pointeur sur pointeur de type double **double ****tab**;**
la variable *tab* est un pointeur pointant sur un pointeur qui pointe sur un flottant double !

La déclaration **int ***p signifie que **p* est un entier et donc que *p* est un **pointeur sur un entier**.

Opérateur adresse (&)

On dispose de deux opérateurs unaires pour manipuler les pointeurs, **&** et *****
L'opérateur d'adresse **&** renvoie l'adresse de l'objet désigné.

```
int i, j;        /* déclarations d'entiers i et j */
int *p1, *p2;    /* déclaration de pointeurs sur des entiers */
i = 5;      /* affectation */
j = 2 * i; /* affectation */
p1 = &i; /* p1 contient l'adresse de i */
p2 = &j; /* p2 pointe sur j */
```

Cet opérateur & ne peut être employé que sur les variables et les éléments de tableaux ; les constantes &3 et les expressions &(i+1) sont interdites.

Opérateur valeur (*)

L'opérateur ***** considère son opérande comme un pointeur et en renvoie le contenu.

p1 = &i; reçoit l'adresse de i; donc pointe sur i

j = *p1; j prend la valeur contenue à l'adresse pointée par p1, donc la valeur de i ;:cette instruction est équivalente à j = i;

```
int *p, *q, x, y;    Partout où un entier peut être employé, *p peut l'être aussi :
p = &x;  /* p pointe sur x */
x = 10;       /* x vaut 10 */
y = *p1 - 1;/* y vaut 9 */
*p += 1; /* incrémente x de 1 : x vaut 11 */
(*p)++;  /* incrémente aussi de 1 la variable pointée par p, donc x vaut 12. */
```

<u>Attention aux parenthèses</u> :

*p++ est TRES DIFFERENT de (*p)++

car ++ et * sont évalués de droite à gauche et *p++ incrémente le pointeur p (l'adresse) et non le contenu de p

```
*p = 0;       /* comme p pointe sur x, maintenant x vaut 0 */
q = p;        /* p et q pointent maintenant sur la même adresse donc *q devient zéro ! */
```

Pointeurs et tableaux

En C le compilateur transforme toute déclaration de tableau en un pointeur sur le premier élément du tableau.

int tabint**[**10**];** est donc équivalent à **int ***tabint**;** (à la réservation mémoire près).

Exemples :

```
int i, j, *p;
p = &tabint[0];    p pointe sur le premier élément de tabint
i = *p;            recopie le contenu de tabint[0] dans i
p++;           p pointe sur le second élément de tabint
p+j;           adresse de tabint[j]
*(p+j);            contenu de tabint[j]
```

Tableaux de pointeurs

Un tableau de pointeurs est un tableau dont les éléments sont des pointeurs.

```
char *mois[13];
int *valeurs[5];
```

Allocation dynamique de mémoire

Quand on utilise des pointeurs il est indispensable de toujours savoir quels éléments sont manipulés dans les expressions et si de l'espace mémoire a été réservé par le compilateur pour stocker les objets pointés. En effet, alors qu'une déclaration de tableau réserve de l'espace mémoire dans l'espace des données pour la variable et ses éléments, celle d'un pointeur ne réserve que la place pour une adresse !

La déclaration **double** x**[**NOMBRE**];** réserve un espace de stockage pour NOMBRE éléments de type réel double et la variable x[0] stocke l'adresse de l'élément n° zéro (le premier élément).
Par contre la déclaration **double ***x**;** ne réserve que le seul espace destiné au pointeur x, AUCUN espace n'est réservé pour une ou plusieurs variables en double précision.

La réservation d'espace est à la charge du programmeur qui doit le faire de façon *explicite*, en utilisant les fonctions standard **malloc**(), **calloc**(), ou **realloc**().

Ces fonctions sont prototypées dans <stdlib.h> et <alloc.h>

char * malloc(**unsigned** taille); réserve taille octets, sans initialisation de l'espace

char * calloc(**unsigned** nombre, **unsigned** taille); réserve nombre éléments de taille octets chacun ; l'espace est initialisé à 0.

void * realloc(**void** *block, **unsigned** taille); modifie la taille affectée au bloc de mémoire fourni par un précédent appel à malloc() ou calloc().

void free(**void** *block); libére le bloc mémoire pointé par un précédent appel à malloc(), calloc() ou realloc().

Ces fonctions doivent être transtypées pour réserver de l'espace pour des types de données qui sont différents du type **(char *)**. Elles retournent le pointeur NULL (**0**) si l'espace disponible est insuffisant.

Exemples :

```
char *s, *calloc();
s = (char *) calloc(256, sizeof(char)); /* réserve 256 octets initialisés à '\0' */
float *x;
x = (float *) malloc(sizeof(float)); /* réserve un espace pour un réel de type double */
```

L'allocation dynamique de mémoire se fait dans le **tas** (mémoire système dynamique) ; la taille de celui-ci varie en cours d'exécution, et peut n'être pas suffisante pour allouer les variables, provoquant le plantage du programme. Il faut donc tester le retour des fonctions d'allocation et traiter les erreurs.

Exemple de traitement d'allocation dynamique.

```
#define BUFSIZE 100
long *numero;
if (!(numero = (long *) calloc(BUFSIZE, sizeof(long))) )
   /* Si échec à réservation de BUFSIZE emplacement de type long */
 {
   fprintf(stderr, "ERREUR espace mémoire insuffisant !\n");
   exit (1); /* fin anticipée du programme ; code de retour 1 */
 }
else   /* le programme continue */
```

L'espace alloué à une variable peut être libéré par free(), dont l'argument est un pointeur réservé dynamiquement.

```
free(s);
free(x);
free(numero);
```

Programme principal et fonctions

Les fonctions permettent de décomposer un programme en entités restreintes. Une fonction peut se trouver :
- dans le même module de texte (toutes les déclarations et instructions qui la composent s'y trouvent),
- être incluse automatiquement dans le texte du programme (#include)
- ou compilée séparément puis liée (linked) au programme.

Une fonction peut être appelée à partir du programme principal, d'une autre fonction ou d'elle-même (récursivité). Elle peut retourner une valeur (int par défaut) ou ne pas en retourner (déclarée void, assimilable à une procédure Pascal). Une fonction peut posséder ses propres variables ; elle n'a qu'un seul point d'entrée et peut avoir plusieurs points de sortie.

Déclaration de fonction avec prototypage

Le prototypage permet au compilateur de faire une vérification de type au moment de la définition et de l'appel.

```
/* prototype de fonction : entête de déclaration suivi par ';' déclaration avec
les types des arguments entre les parenthèses */
type_retourné nom_fonction (type variable1, type variable2, ...)';'
```

Définition

La définition de la fonction (la liste des instructions) reprend le même texte que pour le prototype sans '**;**'

```
type_retourné nom_fonction (type variable1, type variable2, ...)
{
   /* corps de la fonction */
   /* déclarations de variables locales à la fonction */
   /* initialisations */
   /* instructions; */
   return (valeur_retournée);
}
```

Arguments et valeur retournée.

Les arguments de la fonction sont passés par VALEUR ; une copie provisoire de chaque argument est passée à la fonction qui ne peut donc pas modifier l'argument initial d'une fonction appelante.

Mais si on passe un pointeur (une adresse) à une fonction,celle-ci modifiera ainsi l'argument : c'est un passage par VARIABLE ou par ADRESSE. On obtient le même résultat en passant un tableau à

la fonction, puisque les tableaux sont considérés par le compilateur comme un pointeur sur le premier élément du tableau.

Une fonction déclarée **void** ne retourne rien. La valeur retournée peut être ignorée par la fonction appelante.

Si la fonction ne contient pas de **return( );** la valeur retournée n'a pas de sens particulier.

```
/* Appel de fonction */
#include <stdio.h> /* Entrées/Sorties */

/* prototype */
int demo (int i, double n);

void main(void)  /* Programme principal */
{
   int i; /* Définitions */
   double n;
   i=2; /* Affectations */
   n=10.0;
   printf("APPEL DE FONCTIONS \n");
   printf("Valeurs avant appel\n");
   printf("Variable i=%d\n",i);
   printf("Variable n=%f\n",n);
   if (i)
      printf("Valeur renvoyée par fonction =%d\n",demo(i,n));
   printf("Valeurs après appel\n");
   printf("Variable i=%d\n",i);
   printf("Variable n=%f\n",n);
   printf("\n");
}

int demo (int i, double n) /* définition de fonction */
{
   n = n / i;
   printf("\nValeurs dans la fonction\n");
   printf("  Variable i=%d\n",i);
   printf("  Variable n=%f\n",n);
   return((int)n);
}
```

```
-----------------------------------------------
APPEL DE FONCTIONS
Valeurs avant appel
Variable i=2
Variable n=10.000000

Valeurs dans la fonction
    Variable i=2
    Variable n=5.000000
Valeur renvoyée par fonction =5
Valeurs après appel
Variable i=2
Variable n=10.000000
-----------------------------------------------
```

Remarque :

La fonction `demo(i,n)` fait la division de n par i, retourne le résultat **transtypé** entier mais ne modifie ni la valeur de *n* ni celle de *i*. Le contrôle de i avant la division est indispensable pour éviter une division par zéro.

L'exemple suivant propose une fonction qui modifie (par permutation) les éléments d'un tableau; le tableau est passé à la fonction qui renvoie un tableau modifié.

```
/* Appel de fonction */
```

```
#include "stdio.h" /* Entrées/Sorties */
#define MAXTAB 10  /* Constante */

int echange (int i, int n[]); /* prototype de fonction */

/* définition de la fonction */
int echange (int i, int n[])   /* un tableau en argument */
{
 int j, aux;
 for (i=0, j=MAXTAB-1; i < j; i++, j--)
 {
  printf(" Echange %4d et %4d\n",n[i],n[j]);
  aux = n[i];
  n[i] = n[j];
  n[j] = aux;
 }
 return(i);
}

void main(void)  /* Programme principal */
{
 /* Définitions */
 int i;
 int tabint[MAXTAB] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; /* Affectations */
 printf("APPEL DE FONCTION PAR VARIABLE \n");
 printf("Valeurs du tableau avant appel\n");
 for (i=0; i<MAXTAB; i++)
  printf("%4d%c",tabint[i], (i%5==4 || i==MAXTAB-1) ? '\n' : ' ');

 printf("Nombre d'échanges = %d\n",demo(i,tabint));
 printf("Valeurs après appel\n");
 for (i=0; i<MAXTAB; i++)
  printf("%4d%c",tabint[i], i%5==4 || i==MAXTAB-1) ? '\n' : ' ');
}
```

```
-----------------------------------------------
APPEL DE FONCTION PAR VARIABLE

Valeurs du tableau avant appel
 0 1 2 3 4
 5 6 7 8 9
 Echange 0 et 9
 Echange 1 et 8
 Echange 2 et 7
 Echange 3 et 6
 Echange 4 et 5
Nombre d'échanges = 5
Valeurs après appel
 9 8 7 6 5
 4 3 2 1 0
-----------------------------------------------
```

Pointeurs et arguments de fonctions

En raison de l'appel par valeur, une fonction ne peut pas modifier ses arguments. L'utilisation de pointeurs permet de tourner la difficulté.

Soit par exemple la fonction echange(x,y) qui est **censée** échanger ses arguments :

```
void echange (int x, int y) /* INCORRECT */
{
 int aux;
 aux = y;
 y = x;
 x = aux;
}
```

L'appel de `echange(i,j)` avec i et j entiers n'aura aucun effet car ces paramètres sont passés par valeur et ne sont pas modifiés en dehors de `echange().`

```
void echange (int *x, int *y) /* CORRECT */
{
 int aux;
 aux = *y;
 *y = *x;
 *x = aux;
}
```

Maintenant l'appel de `echange(&i,&j)` modifie les variables i et j car les adresses des paramètres sont passées à la fonction (on parlera de passage par VARIABLE ou par ADRESSE).

Fonctions récursives

Si le texte de la fonction fait appel à la fonction elle-même, la fonction est *récursive*. A chaque appel récursif, l'ensemble des variables locales est sauvegardé dans une pile et restauré au sortir de l'appel récursif. C'est le compilateur qui se charge de gérer la pile ; l'utilisateur doit veiller à ce que les appels récursifs se terminent . Sinon une erreur d'exécution se produit quand la pile des appels est pleine (*stack overflow*).

```
/* version récursive de la fonction factorielle */
#include <stdio.h>
unsigned long fact(unsigned long n); /* factorielle */

void main(void)
{
   unsigned long u;
   printf("Entrez un nombre ");
   scanf("%ld ",&u);
   printf("Factorielle %ld : %ld\n",u, fact(u));
}

unsigned long fact(unsigned long n)
{
   if (n <=1)
      return (1);
   else
      return (n*fact(n-1));
}
```

Quelle est la condition d'arrêt de la récursion ? Combien d'appels récursifs pour (4!) ?

Arguments sur la ligne de commande

Un programme en langage C à une seule entrée : c'est la fonction **main()**, qui doit être unique dans tous les fichiers du programme, et se charge d'appeler les autres fonctions.

On peut transmettre des paramètres à un programme C par le biais des paramètres de cette fonction :

```
void main(int argc, char *argv[])
```

argc est **le nombre d'arguments de la commande + 1**. Si argc > 1, la commande a des arguments.
argv est un pointeur sur un tableau de chaînes de caractères contenant les arguments de la commande :

argv[0] est le nom de la commande.
argv[1] est le premier argument
argv[argc-1] est le dernier argument.

Exemple : Que fait le programme ci-dessous ?

```
#include <stdio.h>
#include <stdlib.h>
char *nom_de_mois(int n)
{
 static char *mois[13] =
  {"erreur", "janvier", "février",
   "mars", "avril", "mai", "juin",
   "juillet", "août", "septembre"
   "octobre", "novembre", "décembre"
  };
 return  ((n < 1 || n > 12) ? mois[0] : mois[n]);
}

main(int argc, char *argv[])
{
 char *nom_de_mois();
 if (argc > 1)
    printf( "<%s>\n", nom_de_mois(atoi(argv[1])));
 else
    printf("Usage %s numéro\n", argv[0]);
}
```

Structure

Une structure est une collection de données regroupées sous un seul nom.

Déclaration

Le type hms (heure, minute, seconde)

```
  struct hms {      /* déclaration et nom de la structure */
       int h, m, s; /* champs de la structure */
      } heurelocale; /* nom de variable */
```

Le type date utilise aussi le type hms :

```
  struct date {
    char     *nom_jour,
          nom_mois[4];
    int     jour,
          mois,
          annee;
    struct hms  heure;  /* structure dans une structure */
    };   /* le nom de variable est facultatif */
```

Affectation

L'opérateur . (point) permet d'accéder aux composantes d'une structure.

```
heurelocale.h = 10;
heurelocale.m = 07;
heurelocale.s = 55;

printf("%02d:%02d%:%02d\n",heurelocale.h,heurelocale.m,heurelocale.s) ;
```

L'opérateur -> (Fléche = MoinsSupérieur) permet d'accéder aux composants d'un pointeur sur une structure.

```
struct date * ddnaissance = (struct date) *calloc(1, sizeof(struct date));
     strcpy(ddnaissance->nom_jour, "Lundi");
     strcpy(ddnaissance->nom_mois, "MAR");
     ddnaissance->jour=19;   ddnaissance->mois=3; ddnaissance->annee=1999;
```

Union

Une union est une variable qui peut contenir (à différents moments) des objets de types et de tailles différents.

Exemple :

```
union mot_machine {
    char quatrecar[4];
    int deuxentier[2];
    long unlong;
    } *mot;
```

Cette union, dont l'encombrment mémoire est de 4 octets, peut être initialisée de différentes façons.
Soit octet par octet :

```
  mot->quatrecar[0] = 'a';
  mot->quatrecar[1] = 'b';
  mot->quatrecar[2] = 'c';
  mot->quatrecar[3] = 'd';
```

Soit par séries de deux octest :

```
  mot->deuxentier[0] = 0;
  mot->deuxentier[1] = 1;
```

soit d'un seul coup :

```
  mot->unlong = 1234567;
```

De plus l'espace mémoire désigné par l'union peut être interprété selon chacun de ces aspects par le compilateur...

Directives de compilation

Typedef

Le mot réservé **typedef** crée de nouveaux noms de types de données (synonymes de types déjà définis).
Exemple : `typedef char * STRING;` fait de STRING un synonyme de "char * "

#define

#define et #include sont des commandes du préprocesseur de compilation.
#define est un mot réservé qui permet de définir une *macro*
Exemple :

```
#define ESC 27
```

A la compilation la chaine ESC sera remplacée par 27 dans tout le fichier contenant cette définition de macro.
Attention : une macro n'est pas une instruction donc *il ne faut pas* terminer la définition d'une **macro** par le caractère **';'**

```
#define  sum(A,B)   ((A) + (B))
```

Cette macro remplace tous les appels de `sum()` avec deux paramètres par l'expression somme. Les parenthèses ( ) évitent les erreurs d'association...

#include

```
#include <nom.h>
```

Inclusion du fichier `nom.h` situé dans le répertoire spécifié par l'option de compilation -I en TURBOC.

```
#include "nom.h"
```

Inclusion du fichier `nom.h` cherché dans le répertoire courant puis dans les répertoires spécifiés par l'option de compilation -I en TURBOC.

Directives de compilatio

Les directives de compilation permettent d'inclure ou d'exclure du programme des portions de texte selon 'évaluation de la condition de compilation.

`#if defined (symbole)` /* inclusion si symbole est défini */
`#ifdef (symbole)` /* idem que #if defined */
`#ifndef (symbole)` /* inclusion si symbole non défini */

`#if (condition)` /* inclusion si condition vérifiée */
`#else` /* sinon */
`#elif /*` else if */
`#endif` /* fin de si */
`#undef symbole` /* supprime une définition */

Exemple :

```
#ifndef (BOOL)
#define BOOL char /* type boolean */
#endif

#ifdef (BOOL)
  BOOL FALSE = 0;    /* type boolean */
  BOOL TRUE = 1;    /* définis comme des variables */
#else
  #define FALSE  0  /* définis comme des macros */
  #define TRUE  1
#endif

#if 0
  Cette partie du texte sera ignorée par le compilateur
  cela permet de créer des commentaires imbriqués et d'écarter momentanément du
texte source des parties deprogramme.
#endif
```

Les entrées/ sorties et les fichiers

Pour des questions de portabilité du langage C, les entrées sorties (affichage ; saisie de caractères ; lecture/écriture de fichiers) qui sont tout à fait dépendantes du matériel, n'ont pas été inclusent dans le langage... Il n'y a pas de mots réservés pour les réaliser.
Cependant chaque compilateur fournit une bibliothèque de fonctions standard réalisant les principales entrées sorties. Cette bibliothèque est désignée sous le nom de fichier `stdio.h`
Tout fichier source qui fait référence à une fonction de la bibliothèque standard doit donc contenir la ligne `#include <stdio.h>` en début de fichier.

Un façon élégante de considèrer les entrées / sorties - c'est-à-dire le clavier, l'écran, les mémoires de masse - c'est de les traiter soit comme des fichiers séquentiels (clavier, écran) soit comme des fichiers à accès direct (la mémoire). Le langage C fournit pour chaque type de fichier une collection de fonctions réalisant les opérations élémentaires de lecture et d'écriture.
Ces entrées / sorties ont pour support des flux (de caractères, de blocs, etc.). Un flux est représenté de façon logique par un type `FILE` du fichier `stdio.h`.
`FILE` est un nom pour une structure dont les champs (tampon, caractère courant, nom du flux...) sont utilisés par les fonctions d'entrée sortie [3].
Trois flux sont prédéfinis : stdin, stdout, stderr, qui sont respectiveemnt l'entrée standard, la sortie standard et la sortie erreur standard. A ces flux sont associés par le système d'exploitation, et ceci indépendamment du programme, des fichiers ou des périphériques au moment du lancement du programme. Les affectations par défaut sont le clavier (stdin) et l'écran (stdout et sdterr), mais elles peuvent être modifiées par des redirections.

Fonctions d'entrées / sorties

Les principales fonctions d'entrées / sorties [input / output] concernent la lecture des caractères au clavier et l'écriture des caractères à l'écran.

[3] Je ne saurais trop conseiller à l'apprenti programmeur l'excellent ouvrage de Jacquelin Charbonnel "Langage C, les finesses d'un langage redoutable" Collection U Informatique - Armand Colin 1992

Les fonctions de lecture `getchar(), gets()` et `scanf()` lisent leurs données sur `stdin`. Les fonction `getc(), fgets()` et `fscanf()` leur sont apparentées pour les flux non prédéfinis de type `FILE *`.
Les fonctions d'écriture `putchar(), puts()` et `printf()` écrivent leurs données sur la sortie standard `stdout`. Les fonctions `putc(), fputc()` et `fprintf()` s'adressent à des flux non prédéfinis, de type `FILE *`.
Les fonctions `fread()` et `fwrite()` permettent de lire et d'écrire des blocs d'enregistrements.
La fonction `fseek()` positionne le pointeur de flux à une position donnée (déplacement de la tête de "lecture / écriture") ; la fonction `ftell()` indique la position courante du pointeur de flux.

Lecture du clavier / écriture à l'écran : getchar() et putchar(), gets() et puts()

`getchar()` : lecture (par le processeur) d'un caractère au clavier, "l'entrée standard"
Syntaxe : `int getchar(void);`
Renvoie un entier sur l'entrée standard (cet entier est un code ASCII) ; renvoie EOF quand elle rencontre la fin de fichier ou en cas d'erreur.
`putchar()` : écriture (par le programme) d'un caractère sur l'écran, "la sortie standard"
Syntaxe : `int putchar(int c);`
Place son argument sur la sortie standard. Retourne *c* si la sortie est réalisés, EOF en cas d'erreur.
Les entrées de getchar() et les sorties putchar() peuvent être redirigées vers des fichiers.
`gets()` : lecture d'une chaine de caractères au clavier.
Syntaxe : `char *gets(char *s);`
Lit une chaine depuis l'entrée standard stdin et la place dans la chaîne *s*. La lecture se termine à la réception d'un NEWLINE (saut de ligne) lequel est remplacé dans *s* par un caractère ASCII nul '\0' de fin de chaîne. Renvoie NULL quand elle rencontre la fin de fichier ou en cas d'erreur.
`puts()` : écriture d'une chaine de caractères à l'écran.
Syntaxe : `int puts(const char *s);`
Recopie la chaine s dans la sortie standard stdout et ajoute un saut de ligne. Renvoie le dernier caractère écrit ou EOF en cas d'erreur.

Formatage des entrées clavier / sorties écran : printf() et scanf()

Les fonctions de la famille de `printf()` et de `scanf()` permettent les sorties et les entrées formatées de chaînes de caractères. Un tableau de leurs caractéristiques assez complexes est donné en annexe.

Fichiers de données

Il s'agit de fichiers non prédéfinis, par exemple des données à traiter par programme.
On distingue selon le type de support et de contenu

- les fichier séquentiels : séquence d'enregistrements auxquels on accéde dans l'ordre séquentiel, c'est-à-dire du premier au dernier en passant par tous les enregistrements intermédiaires. Par exemple une bande magnétique (K7 audio). Le clavier est vu par l'ordinateur comme un fichier séquentiel en lecture. L'écran est un fichier en écriture (pas nécessairement séquentiel).
- les fichier à accès direct : séquence d'enregistrements sur mémoire externe ou interne auxquels on accéde directement, par l'adresse individuelle de chaque enregistrement (mémoire vive RAM : Random Access Memory) ou par indirection (tableau indexé).
- les fichier binaires : ce sont des fichiers structurés sous forme d'enregistrements (struct en langage C), dont le contenu doit être interprété par le programme qui les a créés.
- les fichiers textes : ce sont des documents textuels en ASCII, qu'une commande comme TYPE peut afficher.

Opérations sur les fichiers

Un fichier est caractérisé par un nom et une adresse logique attribués par le propriétaire du fichier et convertis en adresse physique absolue dans le système de fichiers de l'ordinateur (rôle du système d'exploitation [Operating System]).

Les opération de base sur les fichiers sont :
- création du fichier (attribution d'un nom et d'une adresse logique / physique) ;
- ouverture du fichier en écriture, en lecture, en mode ajout ;
- lecture d'un enregistrement ;
- écriture d'un enregistrement ;
- positionnement de la tête de lecture (le tampon) en début, en fin, à une adresse relative au début ou à la fin du fichier ;
- fermeture du fichier ;
- suppression du fichier.

L'accès aux enregistrements constitutifs du fichier à traiter se fait au travers d'une fenêtre appelée tampon [buffer] (de lecture ou d'écriture, selon l'opération réalisée), qui est une variable du même type que les enregistrements constitutifs du fichier, ce qui conduit à voir ce tampon comme contenant un élément du fichier.

Descripteurs et flux

La gestion des fichiers étant étroitement dépendante de la plateforme matérielle et logicielle (système d'exploitation), nous ne présenterons ici que ce qui concerne MS-DOS. On dispose de fonctions équivalentes dans les autres systèmes d'exploitation. Deux méthodes de gestions de fichiers sont possibles dans l'envionnement MS-DOS :

- par descripteur de fichier [handler]

- par association au fichier d'un bloc de contrôle [File Control Block] et d'un pointeur de flux [FILE *]

Fonctions C utilisant un descripteur

Ce sont des fonctions de bas niveau. Leur emploi est plutôt destiné à la programmation système.

Fonctions C utilisant un flux

Ces fonctions accédent à des flux, donc manipulent des objets de type FILE - type défini dans le fichier stdio.h.

`fopen()` : ouvre un fichier dont le nom et le mode sont passés en paramètres.

Syntaxe : `FILE *fopen(char * nomfichier, char * mode);`

mode définit les attributs du fichier ; il peut prendre les valeurs

"r" fichier binaire, ouverture en lecture [read] à partir du début
"w" fichier binaire, création et ouverture en écriture [write] (s'il existe le fichier est vidé)
"a" fichier binaire, ouverture en ajout [append] à partir de la fin du fichier
"r+" fichier binaire, ouverture en lecture / écriture à partir du début
"w+" fichier binaire, recréé en lecture / écriture [write] (s'il existe le fichier est vidé)
"a+" fichier binaire, ouverture en lecture / écriture à partir de la fin

"rt" fichier texte, ouverture en lecture [read] à partir du début
"wt" fichier texte, création et ouverture en écriture [write] (s'il existe le fichier est vidé)
"at" fichier texte, ouverture à partir de la fin du fichier
"r+t" fichier texte, ouverture en lecture / écriture à partir du début
"w+t" fichier texte, recréé en lecture / écriture (s'il existe le fichier est vidé)
"a+t" fichier texte, ouverture en lecture / écriture à partir de la fin

Un pointeur sur un tampon de type FILE est créé et renvoyé par la fonction `fopen()` qui retourne NULL si l'opération a échoué.

Exemple :

```
#include <stdio.h>
#include <errno.h>
#include <stddef.h>
#define MAXNOM 40
#define MAXMODE 4
int main(void)
/*    ouverture    d'un
fichier texte en mode
ajout */ {
FILE *fichier;
char nomfichier[MAXNOM] = "biblio.txt";
if ((fichier = fopen(nomfichier, "a+t")) != NULL)
{ /* le fichier est accessible */
  fprintf(fichier,"%s, %d pages\n","Langage C",50);
  fclose(fichier);
}
else
  fprintf(stderr,"Ouverture    de    %s    impossible
\n",nomfichier);
}
```

fwrite() : écriture d'enregistrements dans le fichier

Syntaxe : `int fwrite(type_tampon *tampon, int longueur, int nombre, FILE *fichier);`

le *tampon* a le même type que les enregistrements dans le fichier
la *longueur* est obtenue avec `sizeof(type_tampon)`
nombre est le nombre d'enregistrements écrits à la fois,
fwrite() retourne le nombre d'enregistrements écrits dans le fichier ou -1 en cas d'erreur.

fread() : lire des enregistrements dans un fichier

Syntaxe : `int fread(type_tampon *tampon, int longueur, int nombre, FILE *fichier);`

le *tampon* a le même type que les enregistrements dans le fichier
la *longueur* est obtenue avec `sizeof(type_tampon)`
nombre est le nombre d'enregistrements lus à la fois,
fread() retourne le nombre d'enregistrements lus dans le fichier ou -1 en cas d'erreur.

fclose() : fermeture du fichier

Syntaxe : `int fclose(FILE *fichier);`

fclose() retourne 0, ou -1 en cas d'erreur.

fseek() : déplacement de la tête de lecture sur le fichier

Syntaxe : `int fseek(FILE *fichier, long distance, int position);`

fseek() fait pointer le pointeur associé au fichierà la position située *distance* octets au-delà de l'emplacement *position*.
distance : différence en octets entre *position* d'origine du pointeur fichier et la nouvelle position. En mode texte, distance doit être 0 ou une valeur renvoyée par `ftell()`.
position : une des trois positions d'origine pour le pointeur de fichier (Début : SEEK_SET, Curseur : SEEK_CUR, Fin : SEEK_END)
L'opération suivant un appel à fseek() sur un fichier ouvert en mode mise à jour peut aussi bien être une lecture qu'une écriture.
fseek() renvoie 0 si succés (pointeur repositionné), non 0 si échec.
Voir aussi fgetpos(), fopen(), fsetpos(), ftell(), lseek(), rewind(), setbuf(), tell().

Exemple : Déterminer la taille d'un fichier

```
#include <stdio.h>
long filesize(FILE *f);

int main(void) {
FILE *fichier;
fichier= fopen(
"biblio.txt", "a+t");
if (fichier != NULL) {
 fprintf(fichier,"%s, %d
 pages\n","Langage
 C",50);
 fprintf(fichier,"%s, %d
 pages\n","Algorithmique"
 ,80);
 printf("La taille du fichier %s est %ld octets\n",
 filesize(fichier));
 fclose(fichier);
}
return (0); }

long filesize(FILE *f) {
long curpos, l;
 curpos = ftell(f); /* indique la position courante */
 fseek(f, 0, SEEK_END);  /* fin du fichier */
 l = ftell(f); /* récupére la position en octets */
 fseek(f, curpos, SEEK_SET);  /* replace la tête */
 return (l);
}
```

Formatage des entrées / sorties dans des fichiers : fprintf() et fscanf()

Les fonctions fprintf() et de fscanf() permettent les sorties et les entrées formatées sur les fichiers. Elles s'emploient comme les fonctions printf() et scanf() en indiquant en plus la référence du flux traité.
Exemple : Lecture d'un fichier de températures `temp.dat`

```
/* temp.dat ---
17/11/1997 13
18/11/1997 18
19/11/1997 17
--------------- */
#include <stdio.h>

int main(void)
{
int temp;
char jour[12];
FILE *ft;
```

```
ft=fopen("temp.dat", "rt");
if (ft==NULL)
  return(-1); /* erreur de lecture */
while (! feof(ft)) /* jusqu'à la fin du fichier */ {
    fscanf(ft,"%s %d\n",jour,&temp);
    printf("La température du %s est %d degrés\n", jour,
temp);
}
fclose(ft);
return (0);
}
```

Compilation

La compilation de programmes en C s'effectue en plusieurs étapes.

1°) Précompilation

Le précompilateur supprime les commentaires des fichiers sources, inclut les fichiers #include et substitue les définitions #define dans tout le fichier <Source> (*.C)

2°) Compilation (compile)

Les fichiers sont alors recodés en fichiers <Objet> (binaire) (*.OBJ). Les noms de fonctions sont remplacés par des étiquettes.

3°) Liaison (link)

Les différents fichiers objets sont ensuites "liés", c'est-à-dire que les étiquettes d'appel de fonctions sont remplacées par l'adresse de celles-ci dans leur propre fichier objet. On obtient un fichier *exécutable* (*.EXE).

La compilation est interrompue en cas d'erreur fatale. Les "warnings" sont des erreurs non fatales dont il faut tenir compte pour avoir un programme à peu près fiable et portable.
Mais ce n'est pas parce que la compilation s'est passée sans erreur que le programme est correct. Le compilateur ne vérifie pas les débordements de tableau (Range Check error), ni bien sûr les erreurs d'allocation en mémoire dynamique ou la cohérence du programme.

Interdépendance de fichiers

Il est possible en C de partager un programme en plusieurs fichiers.

Exemple de fichiers interdépendants.

```
TEST.H ==========================
#define TRUE 1
#define FALSE 0
extern void test(char *msg);
  /* la fonction est déclarée extern :
son code de définition est dans un autre fichier */

TEST.C ==========================
/*  Code de la fonction test()
  Peut être inclus dans une bibliothèque (Library) test.lib
  avec l'utilitaire TLIB.EXE ou directement nommé dans le
  fichier projet (en TURBOC) */
```

```
#include <stdio.h>
void test(char *msg)
{
  printf("%s\n",msg);
}

DEMO.C ========================
/* --- Fichier Source du Programme Principal DEMO.C ----
  Programme exécutable appelant la fonction test()
  Doit contenir une fonction main() */
#include <stdio.h>
#include "test.h" /* déclaration de la fonction externe test() */

void main (void)
{
  test("ceci est un test");
}
================================
```

Pour que la compilation se passe correctement, le compilateur doit pouvoir retrouver ses petits...
En TURBOC sous éditeur vous pouvez définir un fichier de PROJET (Alt P) associé à DEMO.C qui indique les dépendances entre les différents fichiers du programme.

```
DEMO.PRJ ========================
demo.c (test.h)
test.c
================================
```

COMPILATION :

de `test.c` avec Alt C : compile to obj

de `demo.c` avec F9

Processus itératifs.

Dans tous les exercices suivants, nous reprenons la même méthode pour introduire de nouvelles notions de programmation : énoncer un problème numérique, en donner un algorithme, le traduire dans les différents langages de programmation à notre disposition.

Exemple : **Ecrire un programme qui affiche les 10 premiers nombres entiers naturels [0, 1, 2, .., 9].**

1. Rappels mathématiques

L'ensemble des entiers naturels IN est infini. Il a un plus petit élément, noté 0 ; tout élément a un successeur. Il n'y a pas de plus grand élément. Les éléments sont ordonnés. IN={0, 1, 2, 3, 4, ..., n, (n+1), ...}

Enoncer l'ensemble des entiers naturels n'est pas une tâche que puisse réaliser un ordinateur, car cet ensemble est infini. Par contre il n'y a pas de difficulté à énoncer un sous-ensemble de IN. Il nous faut cependant introduire une recette de génération des entiers, c'est-à-dire un algorithme. Il est assez légitime de considèrer la construction de IN à partir de son plus petit élément, 0, en appliquant l'axiome "tout élément a un successeur".
Soit le premier élément 0 *Début du processus*
successeur(0) = 0 + 1 == 1
successeur(1) = 1 + 1 == 2
successeur(2) = 2 + 1 == 3
..
successeur(8) = 8 + 1 == 9 *Fin du processus.*

On a mis en évidence un processus répétitif, nous dirons '*itératif*', selon un terme propre à l'informatique. Toute constante en membre droit d'une *affectation* passe en membre gauche à l'itération suivante. Nous devons résoudre deux difficultés. Initialiser le processus, puis l'interrompre quand la borne 9 est atteinte, ce qui implique de compter le nombre d'itérations.

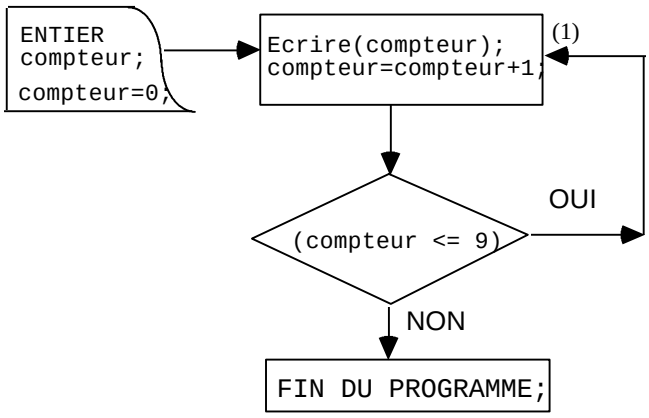
2. Types de données et algorithmes

Définissons les types de données qui seront utilisés.

0 et 1 sont des constantes numériques. Nous utilisons aussi deux opérateurs, l'addition entre entiers et l'affectation. Enfin il nous faut définir un compteur, qui comptera les itérations et permettra d'interrompre le processus. Ce programme pourrait donc se traduire par "Initialiser un compteur à 0. Tant que le compteur est inférieur à 9, incrémenter le compteur de 1".

Traduit de cette façon, le programme est un processus dynamique, c'est-à-dire qui évolue au cours du temps. Le compteur, une variable entiére, prend successivement les valeurs 0, 1, .., 9. L'affichage de la valeur du compteur répond au problème. Il reste à traduire cet algorithme en programme ([4]) ce qui est fait à la figure suivante.

[4] Il faut insister ici sur le caractère très spécial de l'instruction d'affectation qui se trouve dans la boîte (1). Il ne s'agit pas d'une égalité au sens mathématique. On doit plutôt la traduire par : "le contenu de l'adresse mémoire désignée par l'identifieur 'compteur' est augmenté de 1". Pour nous conformer à la syntaxe du langage C nous représenterons dorénavant l'affectation (ou assignation) par le symbole '=', et l'égalite (comparaison) par '=='.



Traduit en langage de la machine RAM cela donne :

ENTIER compteur;
POUR (compteur=0 à 9)
{
  ECRIRE(compteur);
}
FIN;

et en langage C:

```
#include <stdio.h>  /* Fichier inclus : bibliothèque des Entrées/Sorties */

void main(void)  /* Programme principal */
{
int compteur; /* Déclaration */
   for (compteur=0; compteur<10; compteur++)
     printf("%d \n",compteur);
} /* Fin du programme */
```

Nous aurions évidemment pu traduire cet algorithme en utilisant les structures de contrôle TANT QUE (test) FAIRE instruction ; nous en laisserons le soin au lecteur.

Fonctions et sous-programmes

L'exercice suivant va introduire la notion de *fonction* en programmation.

Certains problèmes sont tellement complexes qu'il est préférable de les décomposer en sous-problèmes. Cette forme de programmation, dite programmation descendante, s'attache à exprimer le cadre général de résolution d'un algorithme puis à détailler les sous-programmes résolvant les questions techniques. Donnons un exemple.

Exemple **Ecrire un programme qui affiche les 10 premiers nombres premiers [2, 3, 5, 7,.., 29].**

Comme convenu, commençons par un rappel mathématique. Un nombre *premier* est un nombre entier naturel qui n'est divisible que par 1 et par lui-même (1 n'est pas considéré comme premier).

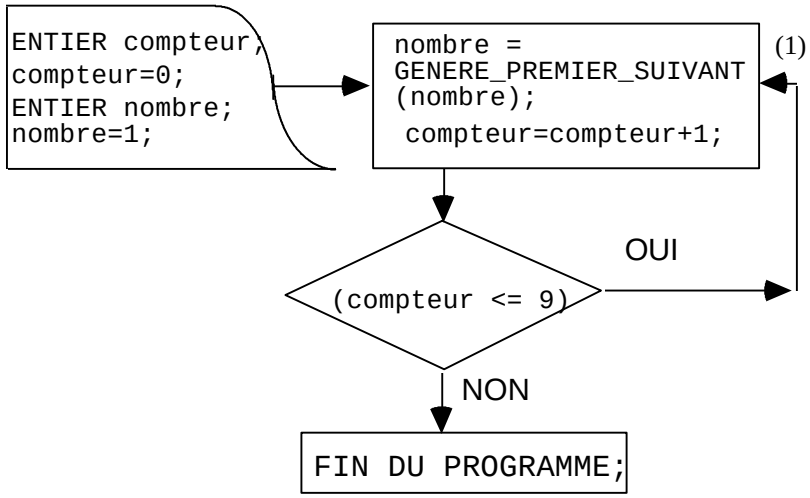
Résoudre l'exercice peut se traduire à première vue par :

```
TANT QUE (compteur<10)
{
  Générer_un_nouvel_entier_premier;
  Incrémenter(compteur);
}
```

Le sous-programme `Générer_un_nouvel_entier_premier;` nécessite d'être formulé plus précisément : étant donné un nombre (éventuellement premier), il s'agit de trouver le nombre premier immédiatement supérieur.

```
nombre=1;
TANT QUE (compteur<10)
{
  nombre = Génére_premier_suivant(nombre);
  Incrémenter(compteur);
}
```

ce qui, traduit en langage graphique, donne

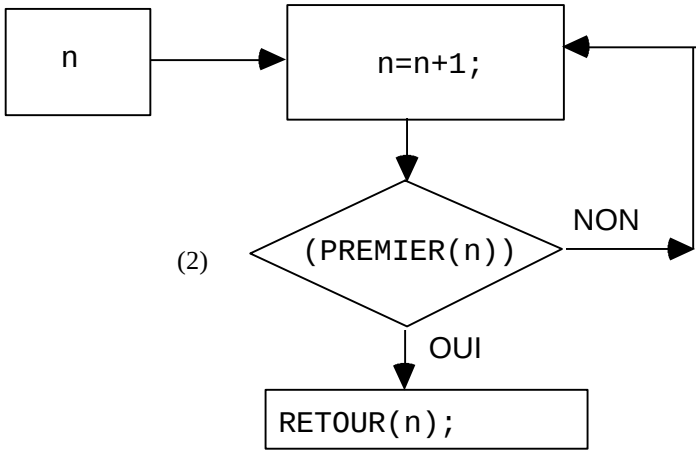


La boucle est triviale, à peu de chose près c'est celle de l'exercice précédent, avec un compteur comptabilisant les nombres premiers générés.
Par contre le *sous-programme* de la boîte (1), générant des nombres premiers, doit maintenant être précisé.

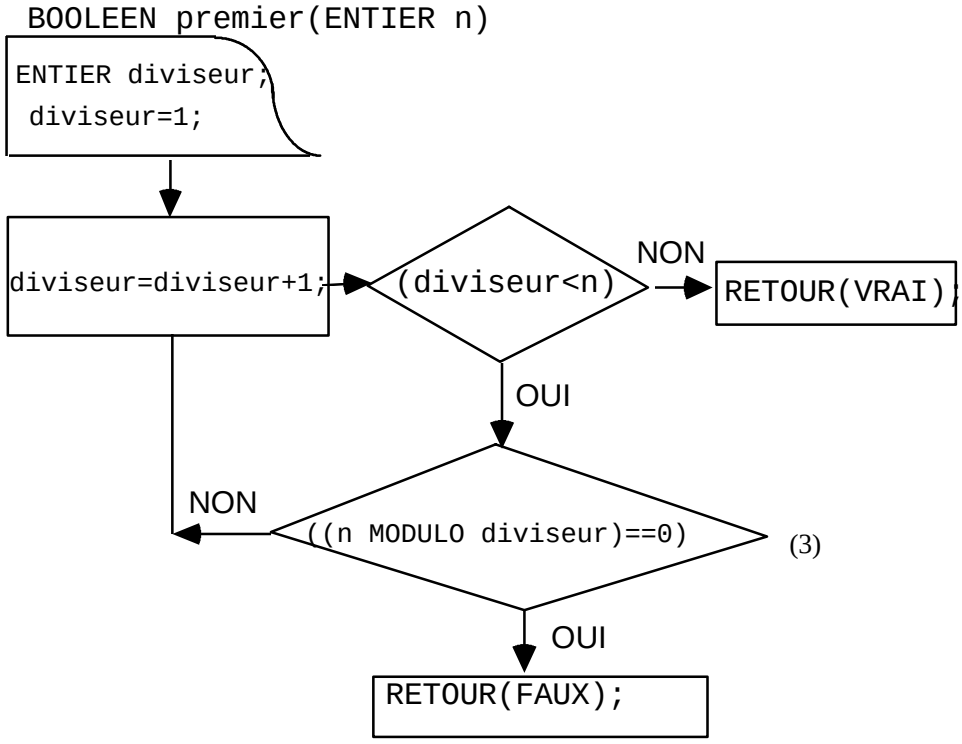
Pour tester si un nombre n'est pas premier, il suffit de lui trouver un diviseur autre que 1 et lui-même. Ainsi à l'exception du nombre entier 2, tous les nombres pairs sont non premiers ; idem des multiples de 3, 5, 7, etc. C'est le crible d'Eratosthéne, qui consiste à éliminer les multiples d'un nombre premier de la liste des nombres entiers. A la "fin" seuls les nombres premiers subsistent. Mais ce procédé ne convient pas à un ordinateur, car l'élimination des nombres pairs est en soi un processus infini. Il faut utiliser une autre méthode, tester par exemple pour chaque nombre s'il est divisible par l'un de ses prédécesseurs inférieurs à n (ou inférieurs ou égaux à n pour gagner quelques itérations)... Cet algorithme n'est pas très efficace, mais il est simple à programmer. Il fait appel à une *fonction* qui retourne VRAI si un nombre est premier, FAUX sinon.

On introduit ici la notion de fonction, qui par construction est un sous-programme retournant une valeur d'un type élémentaire (entier, réel, booléen, caractère, énuméré, pointeur).
Programmons donc la fonction GENERE_PREMIER_SUIVANT(n) qui retourne le nombre premier strictement supérieur à n :

```
ENTIER GENERE_PREMIER_SUIVANT(ENTIER n)
```



Le test (2) fait appel à une fonction, PREMIER(n), qu'il nous faut programmer. Cette fonction retourne VRAI si n est premier et retourne FAUX sinon, elle est donc de type BOOLEEN, c'est pourquoi elle est utilisable dans le test. L'algorithme de cette fonction consiste à rechercher si une division par un entier compris entre 1 et n — 2, 3, etc.— "tombe juste"...



En langage RAM ces fonctions s'écrivent :

```
BOOLEEN premier (ENTIER n)
/* n est un entier strictement supérieur à 1 */
{
   ENTIER diviseur;
   diviseur = 2;
   TANT QUE (diviseur < n)
   {
      SI ((n MODULO diviseur)==0)
         RETOUR (FAUX);
      SINON
         diviseur = diviseur + 1;
   }
   RETOUR(VRAI);
}

ENTIER genere_premier_suivant (ENTIER n)
/* retourne le nombre premier strictement supérieur à n */
{
   n = n + 1;
   TANT QUE ( NON premier(n))
      n= n + 1;
   RETOUR(n);
}

PROGRAMME Liste_10_Premiers_Version1
{
ENTIER nombre, compteur;
nombre=1;
compteur=0;
TANT QUE (compteur < 10)
{
   nombre=genere_premier_suivant(nombre);
   ECRIRE(nombre);
   compteur=compteur+1;
}
```

Ce programme est donc très proche de celui de la génération des dix premiers entiers étudié auparavant. Seule la ligne nombre=GENERE_PREMIER_SUIVANT(nombre);
est modifiée. C'est bien sûr elle qui fait presque tout le travail !
Expliquons en détail sa syntaxe...

GENERE_PREMIER_SUIVANT(*nombre*) est une fonction. Elle reçoit en **paramètre d'entrée** un *nombre* entier qui doit être supérieur à 0. Elle calcule son successeur, (n+1), teste la primalité de ce nombre et retourne ce nombre si le test réussit ; sinon le successeur est testé, jusqu'au succés du test. Le succés est garanti car la suite des nombres premiers n'a pas de majorant... Le nombre ainsi généré est retourné au programme appelant et placé dans la case mémoire désignée par l'identifieur '*nombre*'. L'instruction ECRIRE(*nombre*) affiche ensuite ce résultat.

Simplification du programme

Après avoir minutieusement détaillé ce programme, nous pouvons essayer de le simplifier. Remarquons d'abord que la fonction

GENERE_PREMIER_SUIVANT ()

est une simple boucle qui peut être remplacée par une instruction et un test. Le programme devient :

```
PROGRAMME Liste_10_Premiers_Version2
{
ENTIER nombre, compteur;
nombre=1;
compteur=0;
TANT QUE (compteur < 10)
{
  nombre=nombre + 1;
  SI (premier(nombre))
    ECRIRE(nombre);
  compteur=compteur+1;
}
```

Puis, en considérant qu'au delà de 3, les valeurs successives de n sont des nombres impairs on peut réécrire ce programme :

```
PROGRAMME Liste_10_Premiers_version3
{
ENTIER nombre, compteur;
nombre=3;
compteur=2;
ECRIRE(2);
ECRIRE(3);
TANT QUE (compteur < 10)
{
  nombre=nombre + 2;
  SI (premier(nombre))
    ECRIRE(nombre);
  compteur=compteur+1;
}
```

C'est la version que nous traduisons en Langage C.

```
/* Liste des dix premiers nombres
premiers */
/* Fichier Prem_10.c */

#include <stdio.h>
#define VRAI  1
#define FAUX  0

int premier(int n);
/* n strictement supérieur à 1
   retourne VRAI si n est premier */

void main(void) /* Programme
principal */
{
int compteur, nombre;   /*
Déclarations */
nombre = 3;
compteur = 2;
```

```
printf("10 nombres premiers : 2, 3,
");

while (compteur < 10)
{
   nombre++;
   if (premier(nombre))
   {
     compteur++;
     printf("%d, ",nombre);
   }
}
}    /* Fin du programme */
```

```
int premier(int n) /* assume que n>1
*)
{
   int diviseur=2;
   while (diviseur < n)
   {
     if ((n % diviseur)==0)
       return(FAUX);
     else
       diviseur++;
   }
   return(VRAI);
}
```

Ce programme peut être amélioré en recherchant les diviseurs de n inférieurs à Racine(n). Nous laisserons le lecteur s'y exercer...

Notion de complexité. Amélioration d'un algorithme.

Nous avons observé dans le calcul des nombres premiers qu'un algorithme pouvait être parfois amélioré en tirant partie de certaines propriétés. Cela nous amène a préciser la notion d'efficacité d'un algorithme.

Par définition, on appelle complexité en temps le nombre d'instructions élémentaires mises en oeuvre dans un algorithme. Une instruction élémentaire sera une affectation, un comparaison, un saut. L'évaluation d'une expression, par exemple, "nombre = indice * 2 + 1;" est décomposable en une multiplication, une addition et une assignation, soit trois instructions. Mais comme le décompte précis de toutes les instructions d'un programme risque d'être assez pénible, et qu'entre deux exécutions du même algorithme avec un jeu de paramètres différent, le nombre d'instructions exécutées peut changer, on se contentera en général d'apprécier un ordre de grandeur de ce nombre d'instructions. C'est ce qu'on désignera sous le terme de *complexité en temps de l'algorithme*. (deux instructions élémentaires étant supposées avoir la même durée).

Exemple : Programmer une fonction qui retourne le PGCD de deux entiers naturels non nuls.
Pour trouver le plus grand commun diviseur de deux entiers *a* et *b* non nuls il suffit de lister les *diviseurs de a qui divisent b* et d'en trouver le plus grand. Par exemple pour a=12 et b=18 on trouve D12= {*1*, *2*, *3*, 4, ***6***, 12} ; D18= {*1*, *2*, *3*, ***6,*** 9, 18} soit le pgcd(12,18)=***6***.
Programmons cet algorithme.

```
ENTIER pgc_diviseur (ENTIER a, ENTIER b)
/* on considère que a>0; b>0; a<=b */
{ENTIER diviseur , pgcd;
POUR (diviseur=1 A a) FAIRE
  SI ((a MODULO diviseur)==0)
    ALORS SI ((b MODULO diviseur)==0)
      ALORS pgcd = diviseur;
RETOUR(pgcd);
}
```

Evaluons la complexité de cet algorithme sur un jeu de données. Les cas à considèrer sont :
a et b premiers entre eux ;
a diviseur de b ;
a et b non premiers entre eux sans qu'aucun ne soit diviseur de l'autre ;
Notons entre parenthèses () le nombre d'instructions. Un test sur le modulo compte pour 3 car il faut l'évaluer, comparer sa valeur à 0 et faire un saut.

a	b	diviseur	(a% div.)	(b% div.)	pgcd		Instructions
3	4	1 (1)	0 (3)	0 (3)	1 (1)		(8)
		2 (1)	1 (3)				(4)
		3 (1)	0 (3)	1 (3)		RETOUR 1	(5)
						(1)	**Total (17)**
a	b	diviseur	(a% div.)	(b% div.)	pgcd		
3	6	1 (1)	0 (3)	0 (3)	1 (1)		(8)
		2 (1)	1 (3)				(4)
		3 (1)	0 (3)	0 (3)	3 (1)	RETOUR 3	(8)
						(1)	**Total (20)**
a	b	diviseur	(a% div.)	(b% div.)	pgcd		
4	6	1 (1)	0 (3)	0 (3)	1 (1)		(8)
		2 (1)	0 (3)	0 (3)	2 (1)		(4)
		3 (1)	1 (3)				(4)
		4 (1)	0 (3)	1 (3)		RETOUR 2	(8)
						(1)	**Total (24)**

On peut aussi envisager les cas limites :

a=1 et b>1 : le pgcd retourné est 1 après 8 instructions.

a=b : le pgcd retourné est a, et le nombre d'instructions est de l'ordre de a*6, si on considère qu'en moyenne il y a 6 instructions par ligne du tableau.

On voit sur cet exemple que la complexité de l'algorithme est proportionnelle au nombre de boucles effectuées dans l'instruction "POUR (diviseur=1 A *a*) FAIRE ..." qui dépend du plus petit des deux nombres a et b, chaque boucle coûtant entre 4 et 8 instructions, selon le résultat du modulo. On dira que cet algorithme est en **O(n)** -prononcer "GRAND O de n"- c'est-à-dire d'une complexité linéaire, avec n le plus petit des deux entiers considérés.

Amélioration de l'algorithme.

Considérant que le PGCD est le plus grand des diviseurs communs il suffit de renverser l'ordre de parcours de la boucle et de s'arrêter dés qu'un diviseur commun est trouvé. L'algorithme devient :

```
ENTIER pgc_diviseur (ENTIER a, ENTIER b)
/* on considère que a>0; b>0; a<=b */
{ENTIER diviseur = a;

TANT QUE (diviseur>=1) FAIRE
{
  SI ((a MODULO diviseur)==0)
    ALORS SI ((b MODULO diviseur)==0)
      ALORS RETOUR(diviseur);
  diviseur = diviseur -1;
}
}
```

Si a est diviseur de b, cet algorithme produit la réponse au premier tour de boucle ; sinon sa complexité est en moyenne meilleure, sauf pour les nombres premiers entre eux, pour lesquels il n'y a aucun gain.

En conclusion, dans le moins favorable des cas les deux algorithmes sont d'une complexité en O(n).

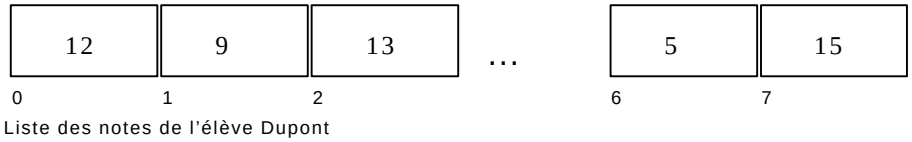
Des données aux structures de données

Le programme de génération des nombres premiers peut être grandement amélioré si plutôt que de rediviser tous les nombres par leurs prédécesseurs on emploie les nombres premiers déjà trouvés comme diviseurs. Autrement dit si dans la fonction **premier**(*nombre*) ce sont des nombres premiers qui sont les diviseurs testés. Cela suppose de ranger en mémoire la liste des nombres premiers au fur et à mesure qu'ils sont générés. Mais la solution qui consisterait à définir au préalable autant de variables que de nombres à trouver n'est ni élégante ni pratique. Un tableau d'entiers est une structure de données bien adaptée à nos besoins.

Tableaux.

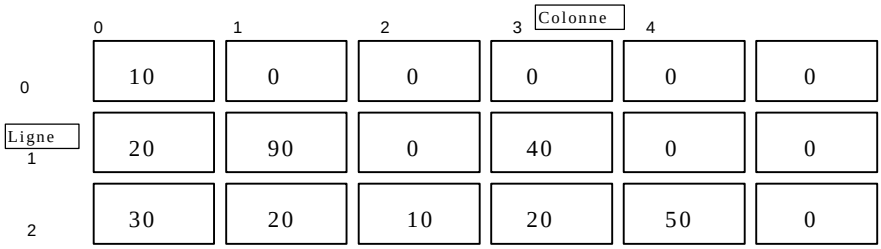
Le terme de tableau a une signification spéciale en informatique. Au sens mathématique usuel du terme, un tableau est une matrice de nombres, c'est-à-dire un objet à deux dimensions (lignes et colonnes), le mot vecteur étant réservé aux objets représentables par une liste de coordonnées, c'est-à-dire la liste des nombres (scalaires) obtenus par projection du vecteur sur chacun des axes de l'espace considéré. En informatique, ce même objet est dénommé tableau à une dimension, une matrice étant assimilée à un tableau à deux dimensions. Et bien sûr la notion peut se généraliser aux tableaux à *d* dimensions. Mais quel que soit le nombre de dimensions, tous les tableaux « informatiques » sont des objets destinés à stocker des données informatiques en mémoire de l'ordinateur. Par définition un tableau de *n* éléments est une collection de *n* cellules mémoires consécutives, de même type (de la même taille). La première cellule du tableau reçoit l'adresse 0, la suivante l'adresse 1, etc. Les *n* cellules sont donc indexées de 0 à *n*-1. L'adressage direct des éléments d'un tableau permet d'accéder en temps constant à chacun d'eux, sans qu'il soit nécessaire de parcourir toutes les cellules précédant celle qu'on veut consulter.

Un tableau à une dimension est un « vecteur »



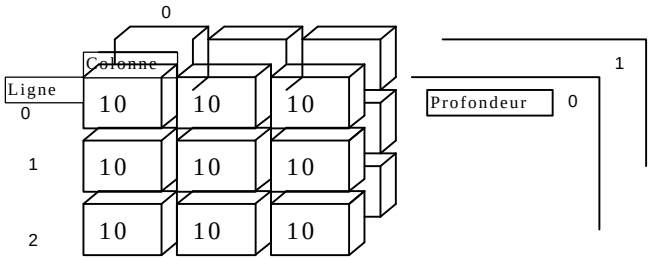
Liste des notes de l'élève Dupont

Un tableau à deux dimensions est une matrice



La cellule d'adresse (1,3) contient la valeur 40.

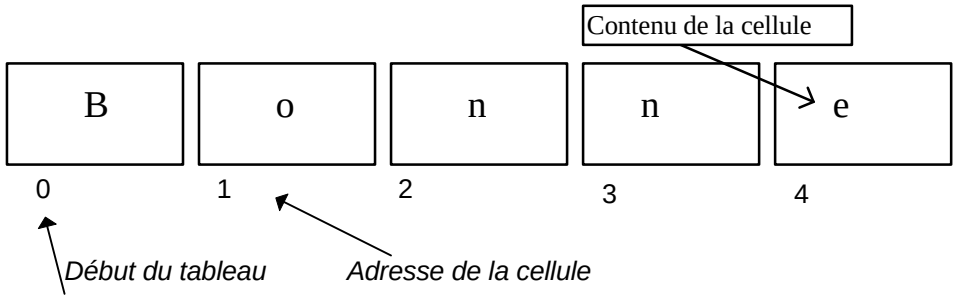
Tableau à trois dimensions



Déclaration d'un tableau.

Avant toute utilisation dans un programme, les structures de données doivent être déclarées au compilateur. Selon le langage de programmation, cette déclaration prend différentes formes. Le principe étant de fournir au compilateur une indication de l'espace mémoire qui sera réservÉ au stockage des données. Nous verrons plus loin que le type d'un tableau n'est pas nécessairement un type simple. Il faut donc distinguer la déclaration du type élémentaire (une cellule) et celle du tableau composé de ces mêmes cellules (qui est aussi le type du tableau).

Un tableau est une suite de cellules consécutives en mémoire



Chaque cellule peut contenir une variable du type de la cellule.
Exemple : En langage RAM, un tableau d'entiers de 100 cellules sera déclaré :

ENTIER **table_entier**[100];

tab1e_entier est un identifieur, dont la syntaxe doit respecter la syntaxe des noms de variables. 100 est nécessairement une constante numérique entiére. ENTIER est un mot réservé caractérisant le type du tableau.
On retrouve une déclaration similaire en langage C :

int table_entier[100];

Il faut insister ici sur le caractère *statique* d'une telle déclaration. En effet il n'est pas possible en cours d'exécution du programme de modifier la taille du tableau, même s'il s'avére que l'espace réservé est insuffisant.

Tableau de tableaux

On a dit qu'un tableau est constitué de cellules contigües en mémoire qui peuvent elles-mêmes être d'un type non simple.
Exemple : **Déclaration d'un tableau de tableaux de réels.**
Soit par exemple à stocker les 10 notes obtenues en informatique par 30 étudiants. Les déclarations suivantes sont équivalentes en espace mémoire :

REEL notes_eleves[30][10];
REEL eleves_notes[10][30];

Dans le premier cas on considère un tableau à deux dimensions de réels constitué par 30 lignes de 10 colonnes ; dans le second cas de 10 lignes de 30 colonnes.

Assignation à un tableau

Assigner une valeur à un tableau c'est placer cette valeur dans une cellule disponible du tableau. Pour cela on dispose du mécanisme d'indexation. Chaque cellule est désignée par son indice, c'est-à-dire son rang à partir de la première cellule, qui elle a l'indice 0 ([5]).

Exemple : Placer les 5 premiers nombres impairs dans un tableau d'entiers.

```
{
ENTIER tab[5];
ENTIER indice;
   POUR (indice=0 à 4) FAIRE
      tab[indice] = (indice * 2) + 1;
}
```

En fin de processus le tableau contient [1, 3, 5, 7, 9]. La première cellule est tab[0] ; elle vaut 1. La cellule tab[3] contient 7.

Exercice

Ecrire en C un programme qui range les 100 premiers nombres premiers dans un tableau.

Algorithme : Initialiser le tableau à 2, 3, 5, 7 puis générer la suite des nombres entiers impairs, en testant pour chaque nombre s'il est divisible par l'un de ses prédécesseurs premiers rangés dans le tableau et inférieur ou égal à Racine(n) (justifiez d'abord cet algorithme).

Corrigé

```
/* Nombres premiers */
#include <stdio.h>
#include <math.h> /* F. Mathéma. */
#define MAXTAB 100

void main(void)         /* Programme
principal */
{
int i=4, j, n=9;
int premier[MAXTAB]={2,3,5,7,}; /*
Déclarations */
printf("2, 3, 5, 7,");
do
{
  j=0;
  while ((n%premier[j]) &&
(premier[j]<=sqrt(n)) && (j<i))
    j++;
  if (n%premier[j])
  {
    printf(" %d,",n);
    premier[i++]=n;
  }
  n+=2;
} while (i<MAXTAB);

}    /* Fin du programme */
```

[5] Nous conserverons la convention du langage C qui est de numéroter 0 la première cellule.Par conséquent pour un tableau de 10 cellules, la dernière a l'indice 9.

Calcul matriciel ([6])

Une matrice réelle est un tableau à deux dimensions M x N (M lignes et N colonnes) d'éléments de type flottant. Un vecteur réel est une matrice réelle à une dimension Nx1 (vecteur colonne de N lignes et 1 colonne) ou 1xN (vecteur ligne de 1 ligne et N colonnes). Une matrice carrée réelle d'ordre N est un tableau NxN à deux dimensions, dont le nombre de lignes et de colonnes sont égaux à N.

Un certain nombre d'opérations algébriques sont définies sur les matrices, selon les tailles respectives des lignes et des colonnes :

- somme et différence de matrices de même dimension et de même taille ;
- produit d'une matrice MxN par un vecteur colonne Nx1 ;
- produit d'un vecteur ligne 1xN par une matrice NxM ;
- produit d'une matrice MxN par une matrice NxP ;
- triangularisation d'une matrice ;
- diagonalisation ;
- transposition ;
- inversion,
- puissance d'une matrice carrée ;
- calcul du déterminant d'une matrice.

Le calcul matriciel ayant énormément d'applications importantes, nous présentons quelques algorithmes et des représentations informatiques de matrices.

Opérations sur les matrices

Type de données matrice

Une matrice NxM est un tableau de flottants à deux dimensions

```
#define MAXLIGNE 5
#define MAXCOL 3
typedef float mat_MxN[MAXLIG][MAXCOL];
```

Une matrice carrée d'ordre N est un tableau de flottants

```
#define ORDRE 5
typedef float mat_carree[ORDRE][ORDRE];
```

Un vecteur ligne est une matrice à une dimension

```
#define MAXELEMENT 10
typedef float vecteur_1xN[MAXELEMENT];
```

Un vecteur colonne est aussi une matrice à une dimension, mais ce sont les éléments d'une colonne qui sont rangés dans le tableau !

On peut aussi transformer les matrices NxM en un tableau à une dimension, de taille N*M, structure bien adaptée pour représenter aussi bien les matrices MxN que NxM, mais avec l'inconvénient important d'avoir à gérer l'adressage dans le tableau, m[i][j] devant alors s'écrire m[i*MAXCOL+j].

```
#define MAXLIGNE 5
#define MAXCOL 3
typedef float m[MAXLIG * MAXCOL];
```

[6] Ce chapitre est repris in extenso de l'excellent ouvrage de Cohen J.H., Joutel F., Cordier Y., Jech B. "*Turbo Pascal, initiation et applications scientifiques*", Ellipses, 1989. J'ai seulement traduit les programmes en C.

On aura intérêt dans tous les cas à définir un type structuré contenant à la fois le tableau des données et la dimension de la matrice.
Par exemple on pourra aussi bien représenter

1.0	**1.2**	**0.5**	0.0
-1.4	**0.9**	**2.1**	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

1.0	**1.2**	**0.5**	**-5.1**
0.0	**0.9**	**2.1**	**0.7**
0.0	0.0	**-3.5**	**4.0**
0.0	0.0	0.0	**1.0**

avec la même structure de données.

```
#define MAXLIGNE 4
#define MAXCOL 4
typedef struct matrice { int l; int c; float m[MAXLIGNE] [MAXCOL]; }t_matrice;
```

Avec cette méthode la première matrice 2x3 est déclarée et initialisée par :

```
matrice mat_2x3 = {2, 3, {1.0, 1.2, 0.5, 0.0, -1.4, 0.9, 2.1, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0}};
```

Bien sûr toutes les valeurs ne sont pas significatives, mais le tableau de données doit être complété avec des valeurs nulles, et c'est par programme qu'on se restreint aux lignes et colonnes utiles...

Produit de deux matrices

Il suffit de trois boucles imbriquées pour calculer le produit de deux matrices.
Avant appel il faut réserver l'espace mémoire pour la matrice produit

```
z = (t_matrice *) calloc(1, sizeof (t_matrice));
```

```
/* Produit de deux matrices */
/* D'après Cohen J.H., Joutel F.,
Cordier Y., Jech B. "Turbo Pascal,
initiation et applications
scientifiques", Ellipses, 1989 */

#include <stdio.h>
#include <conio.h>

#define MAXLIGNE 4
#define MAXCOL 4
typedef struct matrice { int l; int
c; double m[MAXLIGNE] [MAXCOL]; }
t_matrice;

t_matrice unite
     = {3, 3, { 1.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 1.0}};


t_matrice x   = {2, 3, { 1.0, 1.2,
0.5, 0.0,
          -1.4, 0.9, 2.1, 0.0,
           0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0}};

t_matrice y   = {3, 3, { 1.0, 1.4,
0.5, 0.0,
          -1.4, 1.0, 2.0, 0.0,
          -0.5,-2.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0}};

/* prototype */
int produit_matrice (t_matrice *x,
t_matrice *y, t_matrice *z);
void affiche_matrice(t_matrice *x);
```

```
/* sources */

int produit_matrice (t_matrice *x,
t_matrice *y, t_matrice *z)
/* retourne 0 si erreur ; resultat
dans z, dont l'allocation
doit etre faite avant appel */
{
int i, j, k;
if (x->c != y->l)
  return (0);
/* erreur produit impossible */
z->l=x->l;
z->c=y->c;
for (i=0; i < x->l; i++)
  for (k=0; k < y->c; k++)
  {
    z->m[i][k] = 0.0;
    for (j=0; j < x->c; j++)
      z->m[i][k]=z->m[i][k]
+ x->m[i][j] * y->m[j][k];
  }
return (1);
}

void affiche_matrice(t_matrice *x)
{
int i, j;
printf("\t------\n");
for (i=0; i < x->l; i++)
{
    printf("\t");
    for (j=0; j < x->c; j++)
    {
  printf("%2.2f ",x->m[i][j]);
    }
    printf("\n",x->m[i][j]);
}
}
```

```
/* Programme principal */

void main (void)
{
t_matrice *z;

clrscr();
   affiche_matrice(&x);
   affiche_matrice(&y);
   z = (t_matrice *) calloc(1, sizeof
   (t_matrice));

   if (produit_matrice (&x, &y, z))
     affiche_matrice(z);
   if (produit_matrice (&y, &x, z))
     affiche_matrice(z);
   getch();
   }
```

Puissance d'une matrice carrée

On peut utiliser l'algorithme dichotomique récursif de calcul de la puissance entiére d'un réel en l'adaptant au produit de matrices.

Si n=0 x^n = Identité

Sinon

si n est impair $x^n = x \cdot x^{n/2} \cdot x^{n/2}$

si n est pair $x^n = x^{n/2} \cdot x^{n/2}$

Résolution d'un système d'équations (méthode de Cramer)

Pour résoudre un système d'équations linéaires Ax = b, où A est une matrice régulière carrée d'ordre N, et x et b des vecteurs colonnes à N lignes, on transforme le système originel par manipulation de lignes (méthode du *pivot partiel*) ou par manipulation de lignes et de colonnes (méthode du *pivot de Gauss*). Le système d'équations obtenu est Tx = c où T est une matrice triangulaire qui posséde les mêmes solutions que le premier système et se résoud alors facilement par substitutions.

La méthode du pivot partiel

Soient A une matrice carrée et x et b deux vecteurs colonnes.

$$A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ . \\ b_n \end{bmatrix}, \quad \text{et} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ x_n \end{bmatrix}$$

Si a_{11} est non nul, en multipliant la première équation par les coefficients appropriés et en retranchant les équations obtenues aux autres équations du système, on obtient un système équivalent, mais où x_1 n'apparaît plus que dans la première équation:

Eliminons x_1 dans la $i^{éme}$ équation. Multiplions la première équation par (a_{i1}/a_{11}) et retranchons l'équation qui en résulte à cette équation, nous obtenons :

$$a_{i2}^{(1)} * x_2 + a_{i3}^{(1)} * x_3 + \dots + a_{in}^{(1)} * x_n = b_i^{(1)}$$

où

$$a_{ij}^{(1)} = a_{ij} - (a_{i1}/a_{11}) * a_{1j} \quad \text{avec } j \in \{2, \dots, n\} \text{ et } b_i^{(1)} = b_i - (a_{i1}/a_{11}) * b_1$$

Si, enfin, tous les éléments de la première colonne de A sont nuls, la matrice n'est pas régulière et on s'arrête.

Si tout s'est bien passé, on obtient un nouveau système, équivalent au premier : $A^{(1)}x=b^{(1)}$ avec

$$A^{(1)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ . & . & & . \\ 0 & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{bmatrix}$$

$a_{11}^{(1)} \neq 0$ s'appelle le *pivot* de la première élimination.

On recommence alors l'opération sur le système

$$\begin{bmatrix} a_{22}^{(1)} & a_{2n}^{(1)} \\ . & . \\ a_{n2}^{(1)} & a_{nn}^{(1)} \end{bmatrix} * \begin{bmatrix} x_2 \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_2^{(1)} \\ . \\ b_n^{(1)} \end{bmatrix}$$

qui est un système de Cramer, si la matrice A était régulière...

On aboutit en n-1 étapes à un système triangulaire, avec les éléments non nuls sur la diagonale :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & ... & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & ... & a_{2n}^{(2)} \\ . & . & & . \\ 0 & 0 & ... & a_{nn}^{(n-1)} \end{bmatrix} * \begin{bmatrix} x_2 \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ . \\ b_n^{(n-1)} \end{bmatrix}$$

où $a_{ii}^{(i)}$, ($i \in \{1, ..., n-1\}$), sont les pivots successifs des différentes éliminations (le coefficient $a_{nn}^{(n-1)}$ s'appelle aussi un pivot par souci de simplification).

On résout ce système directement pour trouver la solution du système de départ.

$x_n = (b_n^{(n-1)}) / (a_{nn}^{(n-1)})$ et $x_i = (b_i^{(i)} - \sum_{j=i+1 \text{ à } n} a_{ij}^{(i)}) / (a_{ii}^{(i)})$ avec i = n-1, n-2, .., 1

En principe les pivots doivent être simplement choisis parmi les éléments non nuls de la première colonne de la matrice considérée, mais pour des raisons numériques, on prend l'élément qui posséde la plus grande valeur absolue : en effet par exemple en supposant a_{11} non nul et en le chosissant comme pivot, on obtient comme nouveaux éléments de la $i^{éme}$ ligne :

$a_{ij}^{(1)} = a_{ij} - (a_{i1}/a_{11}) * a_{1j}$ avec $j \in \{2, ..., n\}$

Les coefficients de la matrice A et du vecteur b pouvant être des approximations (ce qui sera le cas si A et b proviennent d'une étape numérique antérieure), on obtient (en supposant que $a_{i1} \neq 0$) en ne considérant que les erreurs du premier ordre dues aux approximations :

$\Delta a_{ij}^{(1)} = \Delta a_{ij} - (a_{i1}/a_{11}) \Delta a_{i1} - (a_{i1}/a_{11}) \Delta a_{1j} + (a_{i1} a_{ij}/a^2_{11}) \Delta a_{11}$

$= \Delta a_{ij} + (a_{i1}/a_{11}) (a_{1j} \Delta a_{11}/a_{11} - a_{11} \Delta a_{i1}/a_{i1} - \Delta a_{1j})$

Le terme (a_{i1}/a_{11}) apparaît donc comme un coefficient multiplicatif d'un terme dont un majorant de la valeur absolue est

$|a_{1j}| (|\Delta a_{11}/a_{11}| + |\Delta a_{i1}/a_{i1}|) + |\Delta a_{1j}|$

est symétrique en a_{11} et a_{i1}. En supposant, en outre, que les termes a_{1j} et a_{i1} ($j \in \{2, ..., n\}$) ont des tailles et des incertitudes semblables, on a intérêt à prendre le rapport (a_{i1}/a_{11}) le plus petit possible en valeur absolue (une étude plus approfondie confortant ce choix). De plus, les éléments de la matrice, dans l'une des étapes de l'élimionation, sont obtenues par soustraction. On peut donc craindre que plus les coefficients sont petis, plus les erreurs relatives qui leur sont attachées sont importantes.

Ces deux raisons qui se renforcent mutuellement, justifient le choix du pivot préconisé. Pour améliorer la méthode on peut, c'est la méthode du pivot, chercher le plus grand élément, en valeur absolue, dans la matrice, et permuter les lignes et les colonnes pour l'amener en première position.

Algorithme

Il y a deux étapes : triangularisation de la matrice de départ (méthode du pivot partiel) puis résolution d'un système triangulaire d'équations.

Résolution

```
#define N 5
typedef double matrice_carree[N][N];
typedef double vecteur[N];

void resolution_triangulaire (int ordre, matrice_carree a, vecteur b, vecteur x)
{
```

```
int i, j;
for (i=ordre-1; i>=0; i--)
{
  for (j=i+1; j<ordre; j++)
    b [i] = b [i] - a[i][j] * x [j];
  x[i] =  b[i] / a[i][i];
}
}
```

Triangularisation

La triangularisation de la matrice est une étape préalable à la résolution d'un système d'équations linéaires.

Implantation

```
/* Résolution d'un système d'équations
Méthode du pivot partiel */

#include <stdio.h>
#include <math.h>
#include <conio.h>

#define EPSILON 0.0000001

#define N 3

typedef float matrice[N][N];
typedef float vecteur[N];

matrice unite =
{ 1.0, 0.0, 0.0,
  0.0, 1.0, 0.0,
  0.0, 0.0, 1.0};
matrice a     =
{ 1.0, 2.0, 3.0,
  2.0, 3.0, 1.0,
  3.0, 2.0, 1.0};
vecteur b     =
{ 1.0, 1.0, 1.0};

/* Prototypes */
void affiche_matrice(int ordre,
matrice x);
void affiche_vecteur(int ordre,
vecteur x);
void affiche_systeme(int ordre,
matrice a, vecteur b);
void resolution_triangulaire (int
ordre,
  matrice a, vecteur b, vecteur x);
int gauss_partiel(int ordre, matrice
a,
  vecteur b, vecteur x);

/* Sources */

/* ------------------------------- */
void affiche_matrice(int ordre,
matrice x)
{
int i, j;
printf("\t------\n");
for (i=0; i < ordre; i++)
{
  for (j=0; j < ordre; j++)
    printf("\t%2.3f ",x[i][j]);
  printf("\n");
}
}

/* ------------------------------- */
void affiche_vecteur(int ordre,
vecteur x)
{
int i, j;
  printf("\t------\n");
  for (i=0; i < ordre; i++)
    printf("\t%2.3f ",x[i]);
  printf("\n");
}

/* ------------------------------- */
void affiche_systeme(int ordre,
matrice a, vecteur b)
{
int i, j;
printf("\n");
for (i=0; i < ordre; i++)
{
  for (j=0; j < ordre; j++)
  {
    printf("\t%2.3f * x%d ",a[i][j],
i);
    if (j<ordre-1)printf(" +");
  }
  printf("= %2.3f\n",b[i]);
}
printf("\n");
}


/* ------------------------------- */
void resolution_triangulaire (int
ordre, matrice a, vecteur b, vecteur
x)
{
int i, j;
for (i=ordre-1; i>=0; i--)
{
  for (j=i+1; j<ordre; j++)
    b [i] = b [i] - a[i][j] * x [j];
  x[i] =  b[i] / a[i][i];
}
}


/* ------------------------------- */
int gauss_partiel(int ordre, matrice
a, vecteur b, vecteur x)
```

```
/* triangularise et resout une système
d'équations lineaires ; la taille de
la matrice carree est ordre x ordre ;
Les entrees sont modifiees par effet
de bord !
retourne 0 si erreur */
{
int i, j, k, imax;
float maxi, temp;

for (k=0; k<ordre; k++)
{
  imax = k;
  maxi = fabs(a[k][k]);
/* recherche pivot dans colonne k */
  for (i=k+1; i<ordre; i++)
    if (fabs(a[i][k]) > maxi)
    {
      imax = i;
      maxi = fabs(a[i][k]);
    }
  if (maxi < EPSILON)
  {
    for (i=0; i<ordre; i++)
      x[i] = 0.0 ;
    return (0); /* matrice singuliere
*/
  }
  for (j=k; j<ordre; j++)
/* echange de la ligne k et de la
ligne imax */
  {
    temp = a[k][j]; a[k][j] =
a[imax][j]; a[imax][j] = temp;
  }
  temp = b[k]; b[k] = b[imax]; b[imax]
= temp; /* second membre */
  for (i= k+1; i<ordre; i++)  /*
eliminer dans la colonne k les termes
sous la ligne k */
  {
    temp = a[i][k] / a[k][k];
    for (j=k; j< ordre; j++)
      a[i][j] = a[i][j] - a[k][j] *
temp;
    b[i] = b[i] - b[k] * temp;
  }
}

/* matrice triangulaire superieure */
resolution_triangulaire (ordre, a, b,
x);
return (1);
}

/* ------------------------------- */
/* Programme principal */

void main (void)
{
vecteur x;

clrscr();
printf("Système à résoudre \n");
affiche_systeme(N, a, b);
if (gauss_partiel(N, a, b, x))
{
  printf("Triangularisation\n");
  affiche_matrice(N,a);
  affiche_vecteur(N,b);
  printf("Solution du système \n");
  affiche_vecteur(N, x);
}
getch();
}
```

Calcul du déterminant

La méthode de Gauss permet également de calculer le déterminant et l'inverse d'une matrice carrée régulière d'ordre N. En effet, échanger deux lignes d'une matrice transforme le déterminant en son opposé ; ajouter à une ligne une combinaison linéaire des autres lignes ne modifie pas le déterminant.

Ainsi, lorsque dans la méthode du pivot partiel on obtient une matrice triangulaire, le déterminant de celle-ci est, au signe près, le déterminant de la matrice dont on était parti ; ce signe dépend du nombre d'échanges de lignes fait lors de la recherche des différents pivots : si p est ce nombre le déterminant cherché est le produit de $(-1)^p$ et des n pivots trouvés.

Exercice : Ecrire la fonction `double determinant(int ordre, matrice_carree x)` qui calcule le déterminant d'une matrice carrée d'ordre N par l'algorithme du pivot.

Inversion de matrice

Les manipulations des lignes d'une matrice reviennent à multiplier celle-ci à gauche par des matrices convenables. Il n'est pas difficile de prouver que le produit de ces matrices est la matrice inverse de la matrice de départ lorsque par des manipulations de lignes on transforme cette matrice en l'identité. Pour effectuer ce produit (et en garder la trace) il suffit donc d'appliquer l'ensemble de ces manipulations à la matrice identité : le résultat final est bien ma matrice inverse cherchée.
Le code de la fonction

```
int matrice_inverse(int ordre, matrice_carree a, matrice_carre inv)
```

se déduit donc directement de la fonction `gausspartiel()` :

```
/* Résolution d'un système d'équations
Méthode du pivot partiel */

#include <stdio.h>
#include <math.h>
#include <conio.h>

#define EPSILON 0.0000001

#define N 3

typedef float matrice[N][N];

matrice unite =
   { 1.0, 0.0, 0.0,
     0.0, 1.0, 0.0,
     0.0, 0.0, 1.0};
matrice a     =
   { 1.0, 2.0, 3.0,
     2.0, 3.0, 1.0,
     3.0, 2.0, 1.0};


/* Prototypes */
int produit_matrice (int ordre,
matrice x, matrice y, matrice z);
void affiche_matrice(int ordre,
matrice x);
int matrice_inverse(int ordre, matrice
a, matrice inv);

/* Sources */

/* ------------------------------ */
int produit_matrice (int ordre,
matrice x, matrice y, matrice z)
/* retourne 0 si erreur ; resultat
dans z */
{
int i, j, k;
for (i=0; i < ordre; i++)
   for (k=0; k < ordre; k++)
   {
     z[i][k] = 0.0;
     for (j=0; j < ordre; j++)
       z[i][k] += x[i][j] * y[j][k];
   }
return (1);
}

/* ------------------------------ */
void affiche_matrice(int ordre,
matrice x)
{
int i, j;
printf("\t------\n");
for (i=0; i < ordre; i++)
{
   for (j=0; j < ordre; j++)
     printf("\t%2.3f ",x[i][j]);
   printf("\n");
}
}


/* ------------------------------ */
int matrice_inverse(int ordre, matrice
input, matrice inv)
/* inversion de matrice par la methode
du pivot ; la taille de la matrice
carree est ordre x ordre ; la matrice
entree est prèservée ; retourne 0 si
erreur */
{
int i, j, k, imax;
float maxi, temp;
matrice a;
/* recopie des entrees */
for (i=0; i<ordre; i++)
   for (j=0; j<ordre; j++)
     a[i][j] = input[i][j];

/* matrice inverse */
for (i=0; i<ordre; i++)
   for (j=0; j<ordre; j++)
     inv[i][j] = 0.0;
for (i=0; i<ordre; i++)
   inv[i][i] = 1.0;
/* matrice identite */

for (k=0; k<ordre; k++)
{
   imax = k;
   maxi = a[k][k];
/* recherche pivot dans colonne k */
   for (i=k+1; i<ordre; i++)
     if (fabs(a[i][k]) > fabs(maxi))
     {
       imax = i;
       maxi = a[i][k];
     }
   if (fabs(maxi) < EPSILON)
   {
     return (0);
/* matrice singuliere */
   }
   for (j=0; j<ordre; j++)
/* echange ligne k et ligne imax */
   {
   temp = a[k][j]; a[k][j] =
   a[imax][j];
   a[imax][j] = temp;
   temp = inv[k][j];
   inv[k][j] = inv[imax][j];
   inv[imax][j] = temp;
   }
   for (j=0; j<ordre; j++)
/* normalisation */
   {
     a[k][j] = a[k][j] / maxi;
     inv[k][j] = inv[k][j] / maxi;
   }
   for (i=0; i<ordre; i++)
/* eliminer dans la colonne k les
termes hors diagonale */
   if (i != k)
   {
     temp = a[i][k];
     for (j=0; j< ordre; j++)
     {
       a[i][j] = a[i][j] - a[k][j] *
temp;
```

```
       inv[i][j] = inv[i][j] -
inv[k][j] * temp;
     }
   }
}

return (1);
}

/* ---------------------------- */
/* Programme principal */

void main (void)
{
matrice inv;
matrice z;
int i, j, k;

clrscr();
printf("Matrice à inverser \n");
affiche_matrice(N, a);
if (matrice_inverse(N, a, inv))
{
   printf("Matrice inverse \n");
   affiche_matrice(N,inv);
   printf("Verification \n");
   produit_matrice (N, a, inv, z);
   affiche_matrice(N,z);
}
getch();
}
```

Calcul numérique et programmation de fonctions mathématiques

Jusqu'à présent pour présenter les concepts fondamentaux d'itération, instruction conditionnelle et décomposition en sous-programmes nous avons traité de problèmes d'arithmétiques et d'algèbre. Abordons maintenant quelques notions de calcul numérique.

Cela pourra paraître paradoxal mais l'ordinateur, qui fut conçu comme un calculateur numérique, n'est pas vraiment l'outil idéal pour effectuer des calculs. Ou plutôt l'ordinateur à tout faire d'usage courant n'est pas adapté aux calculs très précis sur les très grands nombres. Cela vient des limitations en espace mémoire et du mode de représentation des nombres en binaire. Mais ce défaut n'est pas propre à l'ordinateur. Par exemple le nombre (1/3) n'a pas de représentation décimale exacte, ce qui ne nous empêche pas de calculer 15 * 0,3333. Bien sûr il vaudrait mieux calculer 15/3 = (5*3) / 3 = 5 * (3/3) = 5... Il faudra donc distinguer entre calculs formels, qui mettent en jeu des expressions algébriques, et calculs numériques, qui utilisent des approximations binaires (retranscrites en décimal à l'affichage).

En ce sens l'ordinateur calculateur est mieux adapté à l'évaluation de grandeurs physiques (éventuellement approchées) qu'au calcul mathématique exact, bien que des programmes comme Mathematica ou Maple soient spécialisés en calcul formel.

Je renvoie le lecteur à son cours d'informatique pour les questions de codage des nombres en binaire. Le programmeur devra toujours avoir à l'esprit (et prendre en compte dans ses programmes) que l'intervalle de définition des entiers sur 2 octets est [-32768 .. 32767] et que les nombres réels ne sont que partiellement représentés par le type FLOTTANT, c'est-à-dire par un couple (mantisse, exposant) de longueurs fixes.

Commençons par un exercice de calcul numérique simple :

Exercice

Ecrire en C un programme qui range 10 nombres réels saisis au clavier dans un tableau, puis en affiche le minimum, le maximum, la moyenne et l'écart-type.

Rappels :

La moyenne (ou espérance) d'une distribution (X_i) s'obtient en calculant

$$\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} X_i$$

L'écart type est la racine carrée de la variance :

$$s = \frac{1}{n}\sqrt{\sum_{i=0}^{n-1}(X_i - \bar{x})^2}$$

Le minimum (respectivement le maximum) d'une distribution est la grandeur X_{min} (X_{max}) qui est inférieure (respectivement supérieure) ou égale à toutes les autres valeurs X_i.

La seule difficulté de l'exercice consiste à définir correctement les structures de données permettant :

1) de stocker la distribution en mémoire

```
REEL tab[10];
```
et en langage C `float tab[10];`

2) restituer les grandeurs demandées.

```
REEL moyenne();     float moyenne();
REEL min();         float min();
REEL max();         float max();
REEL sigma();       float sigma();
```

Le tableau de réels s'impose. Mais le type réel n'étant pas disponible en langage C, il faudra utiliser le type flottant (float). Pour que le programme conserve une certaine généralité, on définira une

constante entiére TAILLE de valeur 10 et on écrira les calculs sous forme de fonctions indépendantes lisant leurs entrées dans la variable globale *tableau[]*, recevant le nombre d'éléments du tableau en paramètre, et retournant un flottant.

Corrigé

```
/* Moyenne, Min, Max, Ecart-type de
10 nombres décimaux rangés dans un
tableau de flottants */

#include <stdio.h>
#include <math.h>

/* variables globales */
#define TAILLE 10
float tableau[TAILLE];

/* macro : carré de deux nombres */
#define CARRE(x) ((x)*(x))

/* prototypes */
float min_tab(int n);
float max_tab(int n);
float moyenne_tab(int n);
float ecart_type_tab(int n);

void main(void) /* Prog. principal */
{
int a, i;   /* variables locales */

printf("Entrez une liste de %d
nombres réels \n", TAILLE);
for (i=0; i<TAILLE; i++)
  scanf("%f",&tableau[i]);

printf("Min : %f, MAX : %f, Moyenne :
%f, Ecart-type : %f\n",
min_tab(TAILLE), max_tab(TAILLE),
moyenne_tab(TAILLE),
ecart_type_tab(TAILLE));

}

float min_tab(int n)
{
int i;
float m = tableau[0];
for (i=1; i<n; i++)
  if (tableau[i]<m)
    m=tableau[i];
return (m);
}

float max_tab(int n)
{
int i;
float m = tableau[0];
for (i=1; i<n; i++)
  if (tableau[i]>m)
    m=tableau[i];
return (m);
}

float moyenne_tab(int n)
{
int i;
float m = 0.0;
for (i=0; i<n; i++)
  m += tableau[i];
return (m/n);
}

float ecart_type_tab(int n)
{
int i;
float m = moyenne_tab(n);
float s=0.0;
for (i=0; i<n; i++)
  s += CARRE(tableau[i] - m);
return (sqrt(s)/n);
}

/* Fin du programme */
```

Remarques

La bibliothèque mathématique est indispensable pour la fonction `sqrt()` qui retourne la racine carrée d'un nombre flottant. La déclaration de constante `#define TAILLE 10` permet de modifier aisément en une ligne la taille du tableau `float tableau[TAILLE].` La définition de macro

`#define CARRE(x) ((x)*(x))` rend la lecture du programme plus claire.

Les fonctions

```
float min_tab(int n);
float max_tab(int n);
float moyenne_tab(int n);
float ecart_type_tab(int n);
```

calculent entre l'indice 0 et l'indice (n-1) du tableau, ce qui permet une certaine souplesse.

Par contre il faudra veiller, si on reprend ces fonctions dans un autre programme, à ce que le paramètre *n* ne soit jamais nul à l'appel de `moyenne_tab (int n)` et `ecart_type_tab (int n)` sous peine d'avoir une division par zéro et l'arrêt brutal du programme.

Calcul des termes d'une suite et problèmes de débordement

Le calcul des termes d'une suite numérique a déjà été abordé pour les nombres premiers. Si une suite est définie de façon récurrente, le programme informatique est trivial. Par contre les problèmes de dépassement de représentation des entiers peuvent rendre le calcul absurde. Voyons cela sur un exemple.

Conjecture polonaise

Ecrire un programme qui teste la conjecture polonaise :

" La suite (u_i) converge vers 1"

avec u_0= n entier strictement positif;

SI (u_i est pair) $u_{i+1}= u_i/2$;

SINON $u_{i+1}= 3u_i + 1$;

Considérons la suite (u_0= 3) ; on obtient la suite (u_i) = (3,10, 5, 16, 8, 4, 2, 1).

Considérons la suite (u_0= 7) ; on obtient la suite (u_i) = (7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Considérons la suite (u_0= 9) ; on obtient la suite (u_i) = (9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1).

Considérons la suite (u_0= 15001) ; on obtient la suite (u_i) = (15001, 45004, ...).

45004 est en dehors de l'intervalle de définition des entiers sur deux octets [-32768.. 32767]. Donc même si l'algorithme est correct, le calcul n'est pas possible. Il faudrait programmer les opérations avec des entiers longs (4 octets), ce qui ne fait que repousser les difficultés à des nombres plus grands. Et l'utilisation de nombres flottants ne résoudrait rien.

Fonctions récursives

Certaines suites ont une définition récurrente qui se prête bien à la programmation dite récursive.

Factorielle

Programmer factorielle(n) notée (n!) définie par :

(0!) = 1; (1!) = 1; (2!) = 2 x (1!); ...; n!= n x ((n-1)!)

Cela se traduit directement en langage C par

```
int factorielle (int n)
{
  if (n<=1)
    return(1);
  else
    return(n * factorielle(n-1));
}
```

Nous dirons que la fonction factorielle est une fonction récursive car sa définition contient un appel à la fonction factorielle elle-même.

Une fonction récursive doit contenir un test d'arrêt de la récursion, sinon elle entre dans une suite infinie d'appels récursifs (en pratique, jusqu'à ce que tout l'espace mémoire disponible pour la pile des appels soit occupé). Le test d'arrêt pour la fonction factorielle est :

```
if (n<=1) return(1);
```

Ce type de programmation est très naturel et nous aurons d'autres occasions de l'employer. Malheureusement ce programme ne peut calculer au delà de (7!), pour des questions de dépassement de représentation. On retombe là dans une situation déjà évoquée.

Exercices de programmation numérique

• Surface et volume d'une sphére

Ecrire un programme qui calcule et affiche la surface et le volume d'une sphére au moyen des formules :

Surface = 4 r ^{2}

Volume = 4 r ^{3} /3

Pour la valeur de on pourra utiliser la fonction `arctan(1)` = /4.

• Polynôme

Evaluer le polynôme suivant :

y = (x-1)/x + 1/2 ((x-1)/x)^2 + 1/3((x-1)/x)^3 + 1/4 ((x-1)/x)^4

Pour simplifier le programme, définir une variable U telle que :

U= (x-1)/x.

• Racine carrée d'un nombre réel positif

On peut calculer la racine carrée de x par la formule itérative suivante :

racine = (x / racine + racine) / 2.

où la première approximation de la racine est égale à 1.

Cette boucle est exécutée jusqu'à ce que la valeur absolue de (x / (racine * racine) -1) devienne inférieure à un ε donné.

Ecrire une fonction racine_carree (x) qui retourne la racine carrée de x par cette méthode.

• Surface d'un triangle

La surface d'un triangle peut se faire de la manière suivante :

Surface = racine carrée de(S (S-a) (S-b) (S-c)), avec S = (a+b+c) / 2 et a, b , c les côtés du triangle.

Ecrire un programme qui calcule et affiche la surface d'un triangle après avoir lu les valeurs a, b et c.

On pourra utiliser la fonction `racine_carree()` définie dans l'exercice précédent.

Cette exercice ne présente aucune difficulté...

• Limite d'une suite

$$S_n = \sum_{i=1}^{i=\infty} \frac{1}{i^2}$$

Calculez la limite de la suite

sachant que l'on arrête l'exécution lorsque $|S_{n+1} - S_n| < e$ avec e un réel proche de zéro.

Commençons par l'analyse de ce problème. Les premiers termes de la suite sont :

$S_1 = \frac{1}{1^2} = 1$; $S_2 = 1 + \frac{1}{2^2} = 1 + \frac{1}{4}$; $S_3 = 1 + \frac{1}{4} + \frac{1}{3^2} = 1 + \frac{1}{4} + \frac{1}{9}$

$S_n = S_{n-1} + \frac{1}{n^2}$

On peut donc mettre en évidence une formule de récurrence : $S_i = S_{i-1} + \frac{1}{i^2}$. Le calcul s'arrête quand $|S_{n+1} - S_n| < e$ c'est-à-dire quand $\frac{1}{n^2} < e$

Programmons cet exercice.

PROGRAMME suiteS(ENTIER n)

```
{
ENTIER i=1;
REEL S=0.0;
REEL EPSILON=0.000001;

TANT QUE ((1/(i*i)>EPSILON) FAIRE
{
    S=S + 1 / (i*i);
    i=i+1;
}
ECRIRE (S);
}
```

- **Valeur approchée de $\pi^2/6$**

Vérifiez expérimentalement que $S_n = \sum_{i=1}^{i=\infty} \frac{1}{i^2}$ est une bonne approximation de $\frac{p^2}{6}$.
Pour la valeur de π on pourra utiliser la fonction `arctan(1)` = $\pi/4$.
Cet exercice est une variante du précédent. Il suffit de tester la différence entre S_n et * /6 jusqu'à ce qu'elle soit inférieure en valeur absolue à un ε donné.

- **Valeur approchée de log(n)**

Calculer à une précision fixée la différence entre la somme $S_n = \sum_{i=1}^{n} \frac{1}{i}$ et Ln(n).
Dans cet exercice, la convergence de S_n vers Ln(n) —logarithme népérien de n— n'est pas du tout garantie. Il faut donc choisir soigneusement le test d'arrêt, par exemple en vérifiant la croissance respective de la série et de la fonction.

- **Termes d'une suite**

Ecrire un programme qui calcule et affiche les valeurs de la suite $(u_n)_{n\ 1}$ (n donné par l'utilisateur) définie par

$$u_n = \sqrt{1+\sqrt{2+\sqrt{3+\sqrt{4+\sqrt{\ldots+\sqrt{n}}}}}}$$

Contrairement à l'exercice précédent, il n'y a pas de formule de récurrence immédiate entre les termes de la suite U_n ; par contre il est facile d'évaluer le n-iéme terme de la suite par une itération avec un invariant de boucle sous le radical :

$U = n$

de $i = n$ à 1 faire

$U = (i - 1) + \sqrt{U}$

Ce qui donne en langage RAM :

```
REEL suiteU(ENTIER n)
{
ENTIER i=n;
REEL U=n;
TANT QUE (i>0) FAIRE
{
  U=(i-1)+√U;
  i=i-1;
}
RETOUR (U);
}
```

```
PROGRAMME ListeUn(ENTIER n)
{
ENTIER i=1;
REEL S=0.0;
REEL EPSILON=0.000001;

POUR (i=1 jusqu'à n) FAIRE
{
  ECRIRE(suiteU(n);
}
```

```
}
```

• **Puissance n-iéme d'un nombre**

Programmer une fonction Puissance(x,n) qui retourne la puissance entiére positive n d'un nombre réel x en tirant partie de la propriété :

$x^0=1$; $x^n=x^p\ x^p$ si $n=2p$ et $x^n=x\ x^p\ x^p$ si $n=2p+1$

Estimer la complexité de l'algorithme.

C'est un exemple de programmation récursive.

```
REEL Puissance (REEL x, ENTIER n)
/* n entier positif ou nul */
{
SI (n==0)
  RETOUR (1);
SINON {
SI ((n MODULO 2) == 0 )
  RETOUR(Puissance(x, n/2) * Puissance(x, n/2));
SINON
  RETOUR(x * Puissance(x, n/2) * Puissance(x, n/2));
}
}
```

Chaque appel récursif divisant le paramètre n par deux, il y a $2*\log_2(n)$ appels, soit une complexité en $O(\log_2(n))$, meilleure que celle du produit x*x..*x qui est en O(n).

• **Nombre d'or**

Soit $f = \frac{1+\sqrt{5}}{2}$ le nombre d'or et $(u_n)_n$ la suite de Fibonacci définie par $u_0 = 0$, $u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$ pour n 2.

Ecrire un programme qui affiche n, u_n et $\frac{f^n}{\sqrt{5}}$

La suite de Fibonacci peut se programmer de façon récursive.

```
ENTIER Fibonacci (ENTIER n)
{
  SI (n==0)
    RETOUR(0);
  SINON SI (n==1)
    RETOUR (1);
  SINON
    RETOUR(Fibonacci(n-2) + Fibonacci(n-1));
}
```

On pourra utiliser la fonction Puissance(x,n) de l'exercice précédent pour calculer $\frac{f^n}{\sqrt{5}}$.

Nous laissons au lecteur le soin de vérifier la convergence des deux suites.

• **Décomposition d'un cube en somme de nombres impairs**

Le mathématicien grec Nikomakhos (1er siècle après JC) écrit dans son *Introduction Arithmétique* que tout cube est égal à la somme de nombres impairs consécutifs. Par exemple :

$1^3 = 1 = 1$

$2^3 = 8 = 3 + 5$

$3^3 = 27 = 7 + 9 + 11$

$4^3 = 64 = 31 + 33$
$= 1 + 3 + 5 + 7 + 9 + 11 + 13$
$= 13 + 15 + 17 + 19$

Ecrire un programme qui lit un entier *n* et donne une décomposition de n^3 (on pourra utiliser une boucle TANT QUE et un BOOLEEN *trouvé* qui indiquera que l'on a trouvé une solution).

Le cas des cubes pairs est trivial, il suffit de décomposer n^3 en ($n^3/2 - 1$) et ($n^3/2 + 1$). Pour les cubes impairs, on peut chercher à décomposer à partir de 1 :

S = (1 + 3 + ... + (2p-1) + (2p+1)) en incrémentant p jusqu'à ce que S n ^{3}...En cas de dépassement, il faut alors éliminer les nombres impairs les plus faibles à gauche :

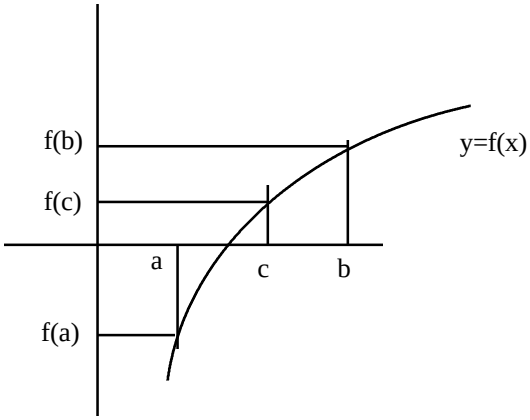
S = ((2q-1) + (2q+1) + ... + (2p-1) + (2p+1)) jusqu'à ce que S n ^{3}. En incrémentant successivement q et p on finit par atteindre n^3 exactement.

Résolution de f(x)=0 ([7])

Soit f une fonction continue dans l'intervalle [a,b] avec a<b et telle que f(a).f(b)<0 ; il existe donc un réel x compris strictement entre a et b vérifiant f(x) = 0. Trouver cette valeur x c'est résoudre l'équation f(x)=0. Plusieurs méthodes numériques s'y emploient. La méthode dichotomique est la plus simple à programmer. Cependant le test d'arrêt doit être soigneusement sélectionné pour éviter les erreurs dues à la représentation approchée des nombres réels par des flottants...

Méthode dichotomique

Considérons c = (a+b) / 2. Si f(c).f(a) 0, il existe une racine de l'équation dans l'intervalle]a, c], sinon il en existe une dans l'intervalle]c, b[. On recommence alors la recherche dans l'intervalle qui contient sûrement une racine, intervalle dont la longueur est la moitié de celle de l'intervalle de départ. En un nombre fini d'étapes, on arrive à encadrer une racine de l'équation par deux réels dont la différence est plus petite que la précision demandée pour la recherche.



c = (a+b)/2.

```
PROGRAMME Dichotomie(REEL a, REEL b, REEL epsilon)
{
REEL c;
SI (f(a)*f(b)) < 0.0
{
   TANT QUE(b-a>epsilon)
   {
      c=(a+b) / 2;
```

[7]J.-H. Cohen, F. Joutel, Y. Cordier, B. Jech, "Turbo Pascal, initiation et applications scientifiques", Elipses, pp127-131

```
      SI (f(a)*f(c)<=0.0)
         b=c;
      SINON
         a=c;
   }
}
}
```

L'inconvénient de cette méthode est que si f(x) est de croissance (respectivement décroissance) lente la précision sur x sera très longue à atteindre. D'autre part il peut se produire que (b/2) et (a/2) soient représentés par le même nombre flottant (la même représentation binaire). En ce cas l'algorithme echoue ! On préférera donc tester une valeur de f(x) proche de zéro à ε près.

Exercice : **Programmer cet algorithme en Langage C pour les fonctions y= x^5/100 et y=2sin(x)-1 sur un intervalle convenable avec une précision de 10^{-5}. Indiquer le nombre d'itérations.**

```
/* Résolution de f(x) = 0 */
/* Méthode Dichotomique */
/* f continue dans l'intervalle [a,b]
et (f(a).f(b)<0
La méthode consiste à calculer c=
(a+b)/2 et f(c) et à remplacer a par
c si f(a).f(c)>=0, b par c sinon. */

 #include <stdio.h>
 #include <math.h>

 #define EPSILON   0.00001
 #define A  -3.0
 #define  B 3.0
 #define  FONCTION(x) (pow((x),5.0) -
1)
/* expression de la fonction f(x) */
 #define  MSG_FONCTION  "x5/100"
/* texte de la fonction */

 int main (void)
 {
  int d=0;
  float a=A, b=B, c;
  float fa, fb, fc;

  printf("Méthode dichotomique
f(x)=%s\n\ta=%2.3f
b=%2.3f\n",MSG_FONCTION,a,b);
  printf("Calcul à EPSILON=%2.6f
près\n",EPSILON);
  fa=FONCTION(a);
  fb=FONCTION(b);
  if ((fa*fb)>=0.0)
  {
    printf("Méthode dichotomique
inapplicable \n");
    return(0);
  }

  do
  {
    d++;
    c = (a+b)/2.0;
    fc= FONCTION(c);
    /* printf("i:%d c:%f
fc:%f",d,c,fc); */
    if ((fa*fc) >= 0.0)
    {
      a = c;
      fa = fc;
    }
    else
      b = c;
  } while (fabs(fc)>EPSILON);
  printf("\n\t%d itérations,
racine:%2.6f \n",d,c);
  return(1);
 }
```

Remarque : Pour rendre le programme plus souple on définit la fonction à traiter sous forme de macro-instructions :

```
#define EPSILON   0.00001
#define A -3.0
#define B 3.0
#define FONCTION(x) (pow((x),5.0) - 1) /* expression de la fonction
f(x) */
#define MSG_FONCTION  "x5/100" /* texte de la fonction */
```

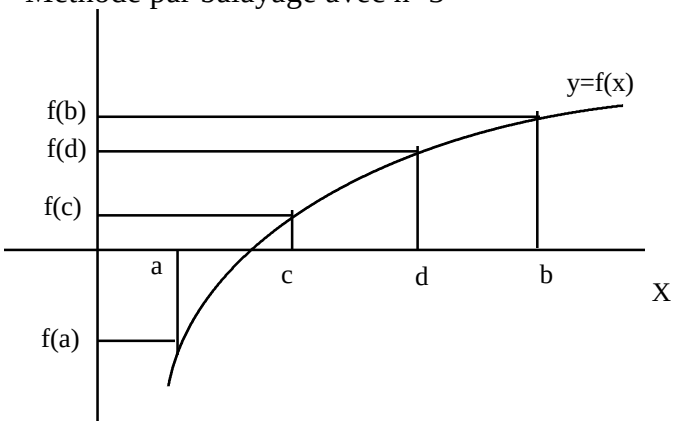
Il suffit de modifier ces cinq ligne de programme pour traiter tout autre fonction.

Méthode du balayage

f continue dans l'intervalle [a,b] et f(a)f(b)<0.

La méthode consiste à déterminer n intervalles sur [a,b], à calculer c = a + k*(b-a)/n et d = a + (k+1)*(b-a)/n et f(c) et f(d) tant que f(c).f(d)>0 et à remplacer a par c et b par d sinon.

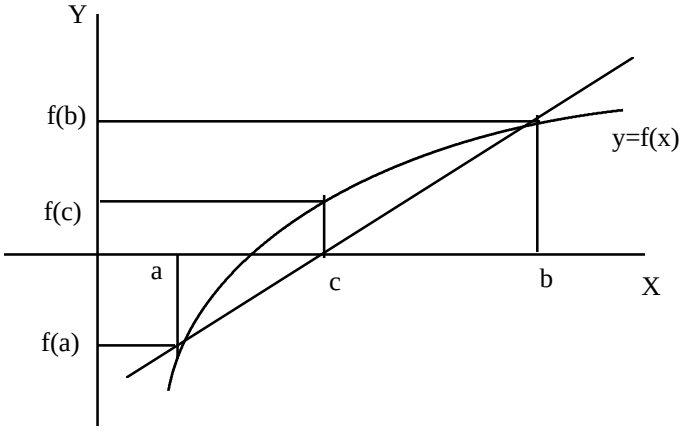
Méthode par balayage avec n=3



Méthode des parties proportionnelles

Les méthodes de recherche par dichotomie convergent lentement. Pour les améliorer on essaie de trouver le point c qui partage l'intervalle de départ le plus près possible d'une racine de l'équation. On trouve que $c = a - f(a)\frac{b-a}{f(b)-f(a)}$ est un bon candidat, égal à la racine si la fonction est affine.

f continue dans l'intervalle [a,b] et f(a)f(b)<0.
La méthode consiste à calculer c, intersection de la droite ((a, f(a)), (b,f(b))) avec l'axe des x ; puis f(c) et à remplacer a par c si f(a).f(c) 0, b par c sinon.



Méthode de Newton

La méthode de Newton abandonne l'idée de l'encadrement, pour essayer d'augmenter la vitesse de convergence. Cette méthode n'assure plus la découverte systématique de la racine, mais lorsqu'elle le fait elle est en général plus rapide que les méthodes précédentes.
Supposons f suffisament dérivable. La formule de Taylor appliquée au point *a* s'écrit :

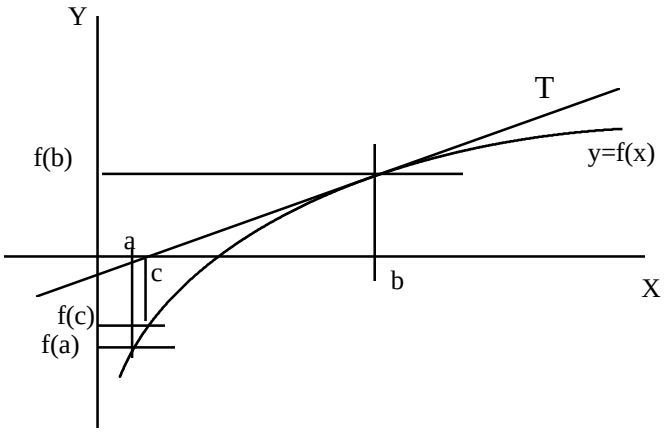
$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2} f''(a+qh),\ q \in]0,1[$$

Si on suppose que *a*+*h* est une racine, *h* est solution de l'équation f(a+*h*) ; en supposant *h* suffisament petit , on en déduit, en première approximation : $h = -\frac{f(a)}{f'(a)}$

On espére ainsi une aproximation d'ordre 2 en *h* de la racine (si f"(*a*+*qh*) est suffisament petit).

Implantation

f continue et dérivable sur l'intervalle [a,b].
Dérivée f' de signe constant (0) sur]a, b[
La méthode consiste à construire la tangente T en B(b, f(b)) et à prendre comme valeur approchée de la racine l'intersection de T avec l'axe des X. b est remplacé par cette valeur et on réitére le processus jusqu'à coïncidence.




```
/* Résolution de f(x) = 0 */
/* Méthode de Newton */
#include <stdio.h>
#include <math.h>

#defineEPSILON0.00001
#defineA    -3.0
#defineB    3.0
#defineFONCTION(x) (pow((x),5.0) - 1)
/* A remplacer par la fonction
calculée */
#defineDERIVEE(x)  (5*pow((x),4.0))

/* A remplacer par dérivée calculée
*/
#defineMSG_FONCTION  "(x^5 - 1)"
/* A remplacer par texte de fonction
*/

/* prototype */
float intersection_tangente_x(float
a, float fa, float dfa);

void main (void)
{
  int d=0;
  float a=A, b=B;
  float fb, dfb;
  printf("Méthode de Newton
f(x)=%s=0\n\t a=%2.3f
b=%2.3f\n",MSG_FONCTION,a,b);
  printf("Calcul à EPSILON=%2.6f
près\n",EPSILON);
  do
  {
    d++;
    fb= FONCTION(b);
    dfb=DERIVEE(b);
    if (dfb != 0.0)
      b =
intersection_tangente_x(b,fb,dfb);
    else
      exit(0);
  } while (fabs(fb)>EPSILON);
  printf("\t%d itérations,
racine:%2.6f \n",d,b);
}

float intersection_tangente_x(float
a, float fa, float dfa)
/* renvoie l'abscisse du point
d'intersection de la droite passant
par[(a, fa)] et de pente dfa, avec
l'axe des x. Assume que dfa != 0 */
{
  return (a - fa / dfa);
}
```

Intégration numérique

Soient a et b deux réels (a<b), *f* une fonction numérique continue sur [a,b] ; on cherche à approximer $\int_a^b f(t)\mathrm{d}t$. Prenons une subdivision régulière de [a,b] en *n* intervalles : posons $h = \frac{b-a}{n}$ et $x_k = a + kh$, pour tout entier *k* compris entre 0 et *n*.

On a :

$$\int_a^b f(t)\,dt = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f(t)\,dt$$

Méthode des rectangles

Appliquons la formule de la moyenne sur l'intervalle $[x_k,x_{k+1}]$, $k \in [1, 2, 3, ..., n-1]$:

$$\exists x_k \in [x_k, x_{k+1}] : \int_{x_k}^{x_{k+1}} f(t)\,dt = (x_{k+1} - x_k) f(x_k)$$

Le réel x_k étant en général inconnu, on cherche une approximation.

La plus simple des approximations que l'on peut faire est de choisir x_k ou x_{k+1}, c'est la méthode communément appelée méthode des rectangles (intéressante quand la fonction est monotone, puisque l'on obtient une borne de la valeur de l'intégrale) ; une autre possibilité est $(x_k+x_{k+1})/2$, qui est le centre de l'intervalle (méthode des rectangles modifiés) et on peut penser a priori que ce choix est meilleur puisque l'incertitude est alors $(x_k-x_{k+1})/2 = h/2$.

On montre qu'un majorant de l'erreur commise, en prenant comme valeur approchée de l'intégrale l'une des valeurs $hf(x_k)$ ou $hf(x_{k+1})$, est $h^2M'/2$, où M' est un majorant de $|f'|$ (en supposant f continûment dérivable, tandis que des méthodes d'analyse plus élaborées montrent que le choix du milieu de l'intervalle conduit à une incertitude de $h^3M''/24$ où M'' est un majorant de $|f''|$ (s'il existe). On voit donc que si la fonction est suffisamment régulière, le choix du milieu est meilleur.

En itérant cette approximation sur chaque subdivision, on obtient alors un majorant de l'erreur totale qui est $M'(b-a)^2/2n$ (méthode des rectangles ordinaires) et $M''(b-a)^3/24n^2$ (méthode des rectangles modifiée).

Méthode des rectangles ordinaires

En prenant, par exemple, systématiquement les bornes gauches des sous-intervalles, l'approximation est :

$$h\sum_{k=0}^{n-1} f(a+kh)$$

Méthode des rectangles modifiée

L'approximation est ici :

$$h\sum_{k=0}^{n-1} f\left(a+\frac{h}{2}+kh\right)$$

Méthode des trapèzes

On approxime la fonction elle-même par une fonction affine sur l'intervalle $[x_k,x_{k+1}]$, $k \in [1, 2, 3, ..., n-1]$ (interpolation linéaire) :

$$f(x) = f_1(x) = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k}(x - x_k) + f(x_k)$$

on pense alors obtenir :

$$\int_{x_k}^{x_{k+1}} f(t)dt = \int_{x_k}^{x_{k+1}} f_1(t)dt = h\ \frac{f(x_k) + f(x_{k+1})}{2}$$

On trouve qu'un majorant de l'erreur faite est $h^3M''/12$ où M'' est une majorant de $|f''|$ (si f est de classe C^2). En itérant l'opération sur tout l'intervalle [a,b], on obtient donc une incertitude totale $M''(b-a)^3/12n^2$.
L'approximation obtenue est donc :

$$h\left(\frac{1}{2} f(a) + \sum_{k=1}^{n-1} f(a+kh) + \frac{1}{2} f(b)\right)$$

Méthode de Simpson

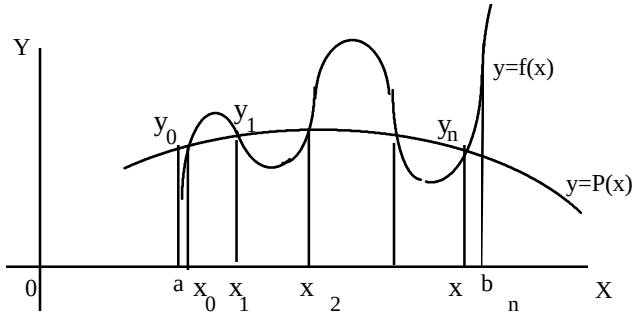
C'est une généralisation de la méthode des trapèzes consistant à approximer la fonction sur chaque sous-intervalle $[x_k,x_{k+1}]$ de la subdivision par un polynôme du second degré (interpolation de Lagrange avec les points x_k, $(x_k+x_{k+1})/2$ et x_{k+1}).

Interpolation ([8])

Supposons qu'en étudiant un certain phénomène, on ait démontré l'existence d'une dépendance fonctionnelle entre des grandeurs *x* et *y* exprimant l'aspect quantitatif de ce phénomène ; la fonction *y*=f(*x*) n'est pas connue, mais on a établi en procédant à une série d'expériences que la fonction *y*=f(*x*) prend respectivement les valeurs y_0, y_1, y_2, y_3, ..., y_n quand on donne à la variable indépendante les valeurs différentes x_0, x_1, x_2, x_3, ..., x_n appartenant au segment [*a*, *b*].
Le problème qui se pose est de trouver une fonction aussi simple que possible (un polynôme par exemple), qui soit l'expression exacte ou approchée de la fonction inconnue *y*=f(*x*) sur le segment [*a*, *b*]. D'une manière plus générale le problème peut être posé comme suit : la valeur de la fonction *y*=f(*x*) est donnée par *n* + 1 points **différents** x_0, x_1, x_2, x_3, ..., x_n du segment [*a*, *b*] :

$y_0 = f(x_0)$, $y_1 = f(x_1)$, ..., $y_n = f(x_n)$;

on demande de trouver un polynôme P(*x*) de degré *n* exprimant d'une manière approchée la fonction f(*x*).
Il est tout naturel de choisir le polynôme de manière qu'il prenne aux points x_0, x_1, x_2, x_3, ..., x_n les valeurs y_0, y_1, y_2, y_3,. .., y_n de la fonction f(*x*).



Ce problème, qui s'appelle "problème d'*interpolation* de la fonction", peut être formulé de la manière suivante : trouver pour une fonction donnée f(*x*) un polynôme P(*x*) de degré *n* qui prenne aux points x_0, x_1, x_2, x_3, ..., x_n les valeurs $y_0 = f(x_0)$, $y_1 = f(x_1)$, ..., $y_n = f(x_n)$.
A cette fin, choisissons un polynôme de degré *n* de la forme

[8] N. Piskounov, "Calcul différentiel et intégral", T1, Editions Mir, pp.264

$$P(x)= C_0(x-x_1)(x-x_2)\ldots(x-x_n)+C_1(x-x_0)(x-x_2)\ldots(x-x_n)+$$
$$+C_2(x-x_0)(x-x_1)(x-x_3)\ldots(x-x_n)+C_n(x-x_1)(x-x_2)\ldots(x-x_{n-1}) \quad (1)_{\text{eet}}$$

déterminons les coefficients $C_0, C_1, \ldots, C_n$ de manière que soient vérifiées les conditions

$$P(x_0)=y_0,\ P(x_1)=y_1,\ \ldots,\ P(x_n)=y_n. \qquad (2)$$

Faisons dans la formule (1) x=x0 ; alors, en vertu des égalités (2), nous avons :

$$y_0 = C_0(x_0-x_1)(x_0-x_2)\ldots(x_0-x_n),$$

d'où

$$C_0 = \frac{y_0}{(x_0-x_1)(x_0-x_2)\ldots(x_0-x_n)}$$

Faisons ensuite x=x1, nous avons :

$$y_1 = C_1(x_1-x_0)(x_1-x_2)\ldots(x_1-x_n)$$

d' où

$$C_1 = \frac{y_1}{(x_1-x_0)(x_1-x_2)\ldots(x_1-x_n)}$$

En procédant de cette manière, nous trouvons successivement

$$C_2 = \frac{y_2}{(x_2-x_0)(x_2-x_1)\ldots(x_2-x_n)}$$

$$\ldots\ldots\ldots\ldots\ldots$$

$$C_n = \frac{y_n}{(x_n-x_0)(x_n-x_1)\ldots(x_n-x_{n-1})}$$

En substituant les valeurs ainsi trouvées des coefficients dans la formule (1) nous avons :

$$P(x)= \frac{(x-x_1)(x-x_2)\ldots(x-x_n)}{(x_0-x_1)(x_0-x_1)\ldots(x_0-x_n)}y_0 + \frac{(x-x_0)(x-x_2)\ldots(x-x_n)}{(x_1-x_0)(x_1-x_2)\ldots(x_1-x_n)}y_1$$
$$+\ldots\ldots+\frac{(x-x_0)(x-x_1)\ldots(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)\ldots(x_n-x_{n-1})}y_n. \qquad (3)$$

Cette formule est appelée formule d'interpolation de Lagrange. Rappelons que si aux points considérés (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), P(x) et f(x) coïncident, en tout autre point de l'intervalle [a, b] il peut en aller tout autrement.
Il existe d'autres formules d'interpolations, dont celle de Newton.

Pour programmer la formule d'interpolation de Lagrange il faut saisir les points (x_0, y_0), (x_1, y_1), ..., (x_n, y_n), dans un tableau de couples de flottants et calculer le tableau des coefficients.

```
/* Interpolation de Lagrange */
#include <stdio.h>

#define MAXPOINT 10

float point2D[MAXPOINT][2]; /*
tableau des coordonnées des points 2D
*/
float coeff_lagrange[MAXPOINT];  /*
coefficients du polynôme */

void saisie_points(int n)
/* assume que les points saisis ont
des abscisses différentes */
{
  int i;
  for (i=0; i<n; i++)
  {
    printf("Abscisse du point N°
%d:",i);
    scanf("%f",&point2D[i][0]);
    printf("Ordonnée du point N°
%d:",i);
    scanf("%f",&point2D[i][1]);
```

```
   }

}

int coefficients_lagrange(int n)
/* le nombre de points est passé en
paramètre */
/* retourne 0 en cas d'erreur */
{
int i,j;
float d_lagrange;
   for (i=0; i<n; i++)
   {
     d_lagrange = 1;
     for (j=0; j<n; j++)
        if (j!=i)
          d_lagrange*=(point2D[i][0]-
point2D[j][0]);
     if (d_lagrange != 0.0)

   coeff_lagrange[i]=point2D[i][1] /
d_lagrange;
     else
        return (0);
   }
   return (1);
}

float affiche_lagrange(int n)
/* Expression littérale du polynôme
*/
{
int i,j;
   printf("P(x)=");
   for (i=0; i<n-1; i++)
   {
     printf("(%f)
",coeff_lagrange[i]);
     for (j=0; j<n; j++)
        if (j!=i)
          printf("(x-
(%f))",point2D[j][0]);
     printf(" +\n");
   }
   printf("(%f) ",coeff_lagrange[n-
1]);
   for (j=0; j<n-1; j++)
     printf("(x-
(%f))",point2D[j][0]);
   printf("\n");
}

float polynome_lagrange(int n, float
x)
/* retourne la valeur du polynôme
pour une valeur x de la variable */
{
int i,j;
float c, p=0.0;
   for (i=0; i<n; i++)
   {
     c=coeff_lagrange[i];
     for (j=0; j<n; j++)
        if (j!=i)
          c*=(x-point2D[j][0]);
     p+=c;
   }
return(p);
}


void main (void)
{
int i,j,n;
float x;
   printf("Interpolation polynômiale
de Lagrange\n");
   printf("Nombre de points à
interpoler [ %d]\n" , MAXPOINT);
   scanf("%d",&n);
   saisie_points(n);
   if (coefficients_lagrange(n))
   {
     /* Expression du polynôme */
     affiche_lagrange(n);
     /* Utilisation du polynôme */
     printf("Entrez une valeur pour
la variable x \n");
     scanf("%f",&x);
     printf("P(%f)=%f",x,
polynome_lagrange(n,x));
   }
   else
     printf("Erreur dans la saisie
des points\n");
}
```

Structures de données

Il est souvent nécessaire en programmation de stocker des informations complexes. Par exemple un répertoire téléphonique. Chaque fiche rassemble les informations caractéristiques d'une entité, en l'occurence une personne, *nom*, *prénom*, *adresse*, *date de naissance*, *téléphone*. Chaque champ d'une fiche est un attribut, exactement défini par son domaine (les valeurs acceptables), sa taille (espace mémoire occupé), et sa valeur (ou occurence). L'information est structurée, puisque chaque fiche est organisée d'une manière figée. Le répertoire téléphonique lui-même est une *collection* de fiches de même structure.

[CHAMP] [TAILLE]	Nom 20 car.	Prénom 10 car.	Adresse 40 car.	Date de nais. date	Téléphone 10 num.
	Dupont	T.	Moulinsart	28/2/1900	12 34 56 78
	Dupond	D.	Moulinsac	24/12/1901	98 76 54 32

Chaque ligne (ou tuple) du tableau à deux dimensions correspond à une fiche.

Structures

Le type du langage C **struct** permet de représenter en machine la fiche que nous venons d'évoquer (en Pascal lui correspond le type **record** — enregistrement).

```
struct fiche {
  char nom[20]; char prenom[10]; char adresse[40];
  char ddn[6]; long tel;
} personne;
```

On peut aussi inclure dans la définition d'une structure la référence à d'autres structures, par une sorte d'imbrication. Par exemple près avoir défini un type de données *date* plus approprié,

```
struct date { int jour; int mois; int annee; } ;
```

s'y référer dans la définition d'un type *fiche*.

```
struct fiche {
  char nom[20]; char prenom[10]; char adresse[40];
  struct date ddn; char tel[10];
} personne;
```

Une *personne* est alors une variable occupant 86 octets en mémoire (20+10+40+6+10).
La mise à jour de la fiche de *Dupond D.* est réalisée par les instructions :

```
strcmpy(personne.nom,"Dupond");
strcmpy(personne.prenom,"D.");
strcmpy(personne.adresse,"Moulinsac");
personne.ddn.jour=24;
personne.ddn.mois=12;
personne.ddn.annee=1901;
strcmpy(personne.tel,"98765432");
```

Nous renvoyons le lecteur à son cours de langage C pour les détails de program-mation.

Exercices

• Le type <Date>

On considère le type <date> défini par une structure C

```
typedef struct date
  {int annee; int mois; int jour;} date;
```

et la déclaration :

```
date date_courante = (1997, 10, 26);
```

1) Quelles opérations sur ce type de données peut-on définir ?

2) Est-il possible d'additionner deux dates ? Que signifierait :
(1994, 10, 26) + (1994, 10, 26) ?
3) La différence de deux dates est une <durée> :

```
long duree; /* exprime le nombre de jours entre deux dates */
duree = difference_date((1994, 10, 26), (1994, 09, 26));
duree = 30; /* en jours */
```

Connaissant la date de naissance d'une personne et la date courante, il est possible d'exprimer l'âge de cette personne en jours :
age = difference_date(date_naissance, date_courante);

Exercice 1: Ecrire en C les fonctions

```
date *saisie_date(void);
void affiche_date(date d);
long difference_date(date d1, date d2);
date *calcule_date(date d, long duree);
date *ajoute_date_duree(date d, long duree);
```

Exercice 2: Ecrire en C un programme qui calcule le calendrier.
Exercice 3 : Définir le type de donnée <heure>

- **Le type <Nombre rationnel>.**

Définition

Les nombres rationnels (ensemble Q) sont définis à partir des entiers relatifs par une relation d'équivalence. Soient deux couples (*a*,*b*) et (*c*,*d*) d'entiers relatifs tels que *b* et d ne soient pas nuls. On appelle nombre rationnel *q*, noté (*a/b*), la classe d'équivalence des couples de nombres entiers relatifs tels que : *a*d = b*c*
Autrement dit on peut assimiler un nombre rationnel à une fraction entiére:

$$\left(\frac{a}{b}\right)=\left(\frac{c}{d}\right)=q\in\mathbf{Q}\Leftrightarrow a\cdot d=c\cdot b \quad \text{avec } a,b,c,d\in\mathbf{Z} \text{ et } b\neq 0 \text{ et } d\neq 0$$

1) Donnez l'équivalent rationnel de deux tiers ; un quart ; 0,5 ; -0,3
2) Le nombre π est-il un rationnel ? Estimez l'erreur commise en prenant pour π la fraction (22/7) ?
3) Quelles sont les opérations autorisées sur les nombres rationnels ?

On définit les opérations suivantes sur les rationnels :
(a/b) * (c/d) = (a*c) / (b*d)
(a/b) / (c/d) = (a*d) / (b*c)
(a/b) + (c/d) = ((a*d) + (b*c)) / (b*d)
(a/b) - (c/d) = ((a*d) - (b*c)) / (b*d)
inverse(a/b)= b/a si a 0, sinon indéfini
Q a une structure d'ordre
Opérateurs de comparaison : > >=< <=
|a/b| |c/d| si et seulement si |a*d| |b*c|
si (a/b) (c/d) alors ((-1)*(a/b)) ((-1) * (c/d))

Implémentation

- Le couple (a,b) —rationnel (a/b)— est défini par une structure en langage C :

```
typedef struct {int n ; int d;} rationnel;
```

Les opérations peuvent être programmées comme des fonctions ; le produit de deux rationnels par exemple sera :

```
rationnel *produit(rationnel x,
rationnel y )
{
rationnel *z;
  z = (rationnel *) calloc(1,
sizeof(rationnel));
  *(z->n) = x.n * y.n;
  *(z->d) = x.d * y.d;
return(z);
```

```
}

rationnel *inverse(rationnel x)
/* retourne un pointeur NULL si
inversion impossible */
{
rationnel *z;
   if (x.n)
   {
      z = (rationnel *) calloc(1,
 sizeof(rationnel));
      *(z->n) = x.d;
      *(z->d) = x.n;
      return(z);
   }
   else
      return(NULL);
}
```

Exercice 1 : Programmer l'égalité, la division, l'addition, la soustraction et la simplification (en utilisant le pgcd) des rationnels.

Exercice 2 : Programmer une calculette en nombres rationnels capable d'évaluer l'expression ((21/3) - (4/8)) / (-24/12) en entrant la séquence :

(21,3) - (4,8) = *13/2*

/ (-24,12) = *-13/4*

Ensembles

La gestion d'informations structurées est un domaine dans lequel les ordinateurs excellent, au prix d'une représentation astucieuse des données. Nous allons passer en revue quelques structures de données classiques.

Définition

Un ensemble est une collection d'objets, appelés éléments de l'ensemble.

Classiquement, un ensemble est défini

• soit en intention, par une propriété vérifiée par tous les éléments de l'ensemble.
Exemples :
- ensemble des entiers naturels impairs
  I = {n / n=2p+1} avec p entier naturel ;
- ensemble des nombre pairs
  M2 = {n / n=2p} ;
- ensemble des multiples de trois
  M3 = {n / n = 3p}
- ensemble des étudiants de Terminale B2 ;
- ensemble des ouvrages de philosophie, etc.

• soit en extention, par l'énumération exhaustive des éléments (ce qui suppose que l'ensemble soit énumérable).
Exemple :
- Arc_en_ciel = {violet, indigo, bleu, vert, jaune, orange, rouge} ;
- Alphabet = {a, b, c, d, e, ...} ;
- Famille Duraton = {Annie, Bernard, Camille, Dominique, Edmond} ;
Cours des monnaies Change = {(Mark, 3.4553), (Ecu, 6.33), (Dollar, 4.89), (FB, 0.1680), ...}

Opérations sur les ensembles

L'algèbre sur les ensembles repose sur quelques opérations élémentaires dont :

• L'appartenance d'un élément à un ensemble (opérateur $\in$)
Exemple : (Lire, 0.00306) $\in$ Change ; Patrick $\notin$ Duraton

• La réunion de deux ensembles, opérateur $\cup$
Exemple : Alphabet = Voyelle $\cup$ Consonne

• L'intersection de deux ensembles, opérateur $\cap$
Exemple : M6 = M2 $\cap$ M3

Rappelons que la notion d'ensemble n'implique pas de relation d'ordre entre les éléments (bien qu'un ensemble puisse être ordonné), et qu'un ensemble n'admet pas de doublons.

Représentation des ensembles : tableaux, listes, piles, files

Il faut distinguer d'une part la représentation en machine des éléments d'un ensemble, et d'autre part l'implémentation des opérations élémentaires (appartenance, union, intersection). Ces opérations dépendront en partie de la représentation des données (les éléments).

• Tableaux

La représentation sous forme de tableau est la plus immédiate. En reprenant le cas d'un répertoire téléphonique, constitué de 200 fiches on peut définir :

```
struct fiche personne[200];
```

Appartenance

Vérifier l'appartenance d'un élément consistera alors à parcourir tout le tableau (soit n comparaisons s'il y a n fiches...)
Il faut disposer d'une fonction permettant de comparer deux éléments, qui doit être adaptée aux données à comparer, mais dont la spécification est en général toujours la même :

si a==b, comparer(a,b) retourne 0
si a<b, comparer(a,b) retourne -1
si a>b, comparer(a,b) retourne +1.

```
#define FAUX 0
#define VRAI 1
int comparer(struct fiche e1, struct fiche e2);
/* retourne -1 si e1<e2 ; 0 si e1==e2; 1 si e1>e2 */

int appartient(struct fiche elt, int nbelt, struct fiche e[])
/* retourne FAUX si l'élément est absent */
{
  int i;
  for (i=0; i<nbelt; i++)
    if (comparer(elt, e[index])==0)
      return(VRAI);
  return(FAUX);
}
```

Réunion

La réunion de deux ensembles peut se concevoir de différentes façons, par exemple en insérant un à un les éléments de l'un des ensembles dans l'autre ensemble.

```
int inserer(struct fiche elt,
     int *p, struct fiche e[])
{
  e[(*p)++]=elt;
}

int union_ensemble (int nbeltA, int nbeltB, struct fiche eA[], struct fiche
eB[])
{
int i;
for (i=0; i<nbeltA; i++)
  if (! appartient(eA[i], nbeltB, eB)
    inserer(eA[i], &nbeltB, eB);
}
```

Intersection

L'intersection de deux ensembles nécessite de créer un nouvel ensemble :

```
int inter_ensemble(int nbeltA, int nbeltB, int *nbelC, struct fiche eA[], struct
fiche eB[], struct fiche eC[])
{
int i;
for (i=0; i<nbelementA; i++)
  if (appartient(eA[i], nbeltB, eB))
    inserer(eA[i], &nbelC, eC);
}
```

Complémentaire

Le complémentaire de A dans B peut se programmer en calculant B-A ; il faut disposer en ce cas d'une fonction de suppression d'un élément d'un ensemble. L'autre méthode consiste à créer un nouvel ensemble C ne contenant que les éléments de B n'appartenant pas à A :

```
int difference_ensemble(int nbeltA, int nbeltB, int *nbelC, struct fiche eA[],
struct fiche eB[], struct fiche eC[])
{
```

```
int i;
for (i=0; i<nbeltB; i++)
  if (! appartient(eB[i], nbeltA, eA))
    inserer(eB[i], &nbelC, eC);
}
```

Complexité en temps des opérations ensemblistes

Les opérations union, intersection, complémentaire utilisent l'opérateur appartenance, qui est en O(n). Pour des ensembles de taille respective *n* et *m* éléments, ces opérations sont donc en O(*n***m*), ce qui est coûteux au delà de quelques dizaines d'éléments...

La représentation des ensembles par des tableaux est pratique pour les ensembles dont la taille est réduite, déterminée a priori, et n'évoluant pas trop en cours d'exécution ; son inconvénient principal tient au coût élevé de la recherche d'un élément, si l'ensemble n'est pas ordonné. Il faut dans le cas contraire envisager d'autres structures de données mieux adaptées. Nous allons en étudier quelques-unes

- **Listes chaînées**

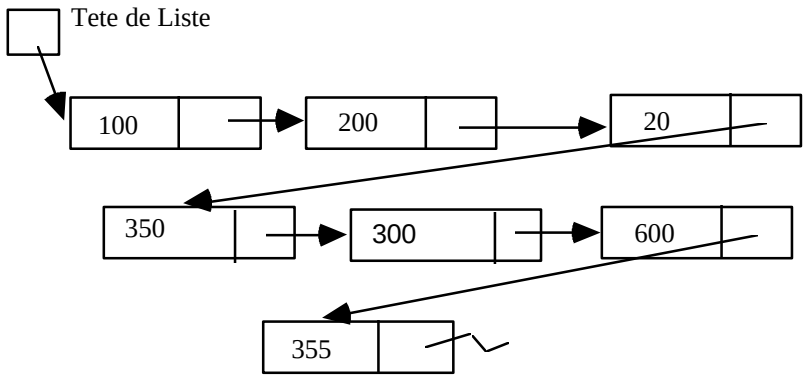
Si la taille de l'ensemble est susceptible d'évoluer de façon dynamique (à l'exécution), ou difficile à déterminer a priori, il est plus judicieux de représenter un ensemble par une liste chainée par pointeur.

Définition

Une liste est une suite (finie) de cellules contenant un élément. Chaque élément a donc une position à l'intérieur de la liste. Tout élément de la liste (sauf le premier et le dernier) a un successeur et un prédécesseur. Un élément peut apparaître plusieurs fois (sauf si la liste implante un ensemble)... En Langage C, on définit le type abstrait **cellule** comme une structure

```
struct  cellule {
         type_donnee val;
         struct   cellule *suiv ; } ;
```

dont le premier champ —val— contient un élément de l'ensemble, et dont le deuxiéme champ —suiv— est un *pointeur* sur la cellule suivante. Un ensemble est une **liste** de cellules chaînées par pointeur comme les perles d'un collier.



L'ordre sur les éléments n'est pas nécessaire pour implanter le type ensemble mais alors la liste ne doit pas contenir de doublon. Du point de vue informatique, l'ajout d'un nouvel élément nécessite la création d'une nouvelle cellule, ce qui se réalise de façon dynamique par allocation de mémoire —fonction malloc() ou calloc()— en cours d'exécution. Par ce procédé la taille de l'ensemble n'est limitée que par l'espace mémoire disponible ; elle n'a pas à être fixée a priori.

Allocation

Considérons par exemple un ensemble de nombres entiers. Une cellule sera définie par :

```
typedef struct  cellule {
          int elt;
          struct   cellule *suiv ; } liste;
```

Une fonction alloc_cellule(int x) d'allocation mémoire d'une cellule retour-nant un pointeur sur une cellule dont la valeur est *x*. sera définie en C par :

```
liste * alloc_cellule (int x)
{
     liste *c;
     c = (liste *) calloc(1, sizeof(liste));
     if (c!=NULL)
     {
           c->elt=x;  /* assignation de x */
           c->suiv = NULL;
     }
     return (c);
}
```

Appartient

Il nous faut écrire une fonction **int appartient(int *x* , liste*l*)**
qui renvoie VRAI si la valeur *x* appartient à la liste *l* , FAUX sinon.
Vérifier si x appartient à la liste, cela consiste à comparer x avec l'élément en tête de liste, puis à parcourir la liste en suivant le chaînage, soit jusqu'à trouver l'élément recherché, soit jusqu'à atteindre la fin de liste. La programmation récursive s'impose.

```
int appartient (int x, liste *l)
{
     if ((l==NULL)
           return (FAUX);
     else if (l->elt==x)
           return (VRAI);
     else
           return (appartient(x, l->suiv));
}
```

Variante de la fonction appartient(), la fonction

liste position_element(int *x* , liste*l*)

renvoie un pointeur sur la cellule contenant *x* s'il se trouve dans la liste, et NULL sinon.

```
liste *position_element(int x, liste *l)
{
     if ((l==NULL)
           return (NULL);
     else if (l->elt==x)
           return (l);
     else
           return (position_element(x, l->suiv));
}
```

Réunion, intersection, complémentaire

Insertion d'un élément

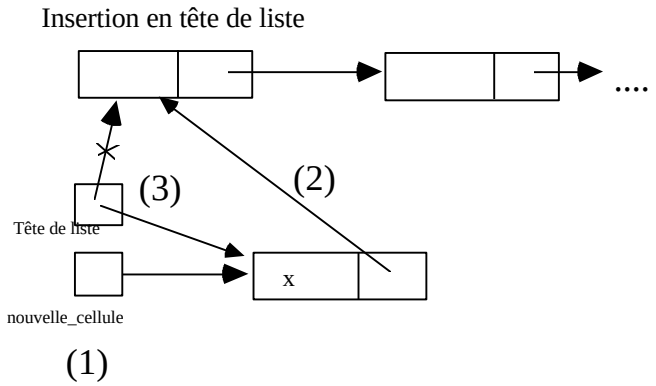
La réunion, l'intersection et le complémentaire nécessitent de disposer d'une fonction d'insertion d'un élément dans une liste. Si aucun ordre n'est imposé, l'insertion en tête est la moins coûteuse.

liste * insere_en_tete(int *x* , liste **l*)

insére un nouvel élément en tête de liste ***l*** et retourne un pointeur sur la liste.

```
liste *insere_en_tete(int x , liste *l )
{
liste *c;
```

```
   c = (liste *) calloc(1, sizeof(liste));
   if (c!=NULL)
   {
     c->elt=x; /* assignation de x */;
     c->suiv=l;  /* assignation de suivant */
     l=c;   /* chainage */
   }
   return (l);
}
```

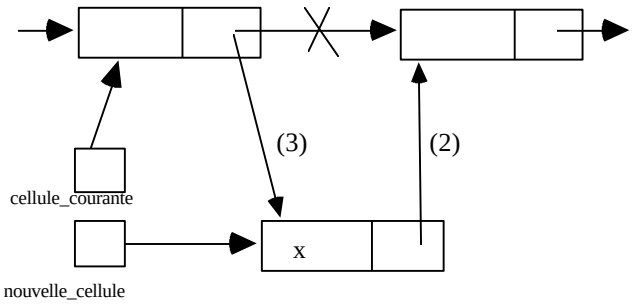


Insertion en tête : (1) créer une cellule ; (2) chaîner avec la première cellule de la liste ; (3) faire pointer la tête de liste vers la nouvelle cellule.

Si l'ordre des éléments doit être respecté, l'insertion dans le corps de la liste s'impose. Il faut alors repérer la cellule précédant immédiatement la nouvelle cellule à insérer, et modifier les liens comme indiqué sur le schéma suivant.

```
liste *insere_en_place(int x , liste *l )
{
liste *c, *aux;
   aux=l;
   /* tester si insere en tête */
   if ((aux==NULL) || ((aux->elt)>x))
     return(insere_en_tete(x,l));
   /* sinon rechercher la position de x */
   while ((aux->suiv)!=NULL) && ((aux->suiv->elt)<x))
     aux=aux->suiv;
/*1*/  c = (liste *) calloc(1, sizeof(liste));
   if (c!=NULL)
   {
     c->elt=x; /* assignation de x */;
/*2*/    c->suiv=aux->suiv; /* assignation de suivant */
/*3*/    aux->suiv=c;  /* chainage */
   }
   return (l);
}
```

Insertion dans le corps de la liste

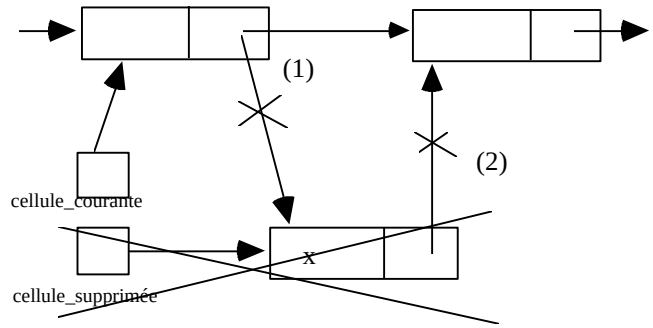


(1)

Suppression d'un élément.

La suppression d'un élément nécessite de l'isoler dans la liste, puis de supprimer le lien de cet élément avec son prédécesseur et son successeur.
Nous laissons cet exercice au lecteur.

Suppression
d'une cellule



(3)

Programmons les opérations Réunion, Intersection et Complémentaire.

```
liste *union_liste(liste *a, liste
*b)
{
liste *aux, *c=NULL;
/* recopie des éléments de a dans c
*/
   aux=a;
   while (aux!=NULL)
   {
     c=insere_en_tete(aux->elt, c);
     aux=aux->suiv;
   }
/* recopie des éléments de b dans c
*/
   aux=b;
   while (aux!=NULL)
   {
     if (! appartient(aux->elt, a))
        insere_en_tete(aux->elt, c);
     aux=aux->suiv;
   }
   return (c);
}
```

```
liste *inter_liste(liste *a, liste
*b)
{
liste *aux, *c=NULL;
/* recopie des éléments de b
appartenant à a dans c */
   aux=b;
   while (aux!=NULL)
   {
     if (appartient(aux->elt, a))
        c=insere_en_tete(aux->elt, c);
     aux=aux->suiv;
   }
   return (c);
}

liste *difference_liste(liste *a,
liste *b)
{
liste *aux, *c=NULL;
/* recopie des éléments de b
n'appartenant pas à a dans c */
   aux=b;
   while (aux!=NULL)
```

```
  {
    if (! appartient(aux->elt, a))
      c=insere_en_tete(aux->elt, c);
    aux=aux->suiv;
  }
  return (c);
}
```

Complexité

La complexité de ces opérations est en O(n*m), avec n (respectivement m) éléments dans A (respectivement B).

Le programme complet suivant génére deux ensembles d'entiers tirés au hasard, et en calcule l'union, l'intersection et la différence.

```
#include <stdio.h>
#include <stdlib.h>

#define VRAI 1
#define FAUX 0


typedef struct   cellule {
     int elt;
     struct  cellule *suiv ; } liste;



liste * alloc_cellule (int x ) ;
int appartient (int x, liste *l);
liste *position_element(int x, liste
*l);
liste *insere_en_tete(int x, liste *l
);

liste *insere_liste(int x, liste *l
);
liste *union_liste(liste *a, liste
*b);
liste *inter_liste(liste *a, liste
*b);
liste *difference_liste(liste *a,
liste *b);
int affiche_liste(liste *a);


liste * alloc_cellule (int x )
{
   liste *c;
   c = (liste *) calloc(1,
sizeof(liste));
   if (c!=NULL)
   {
      c->elt=x;   /* assignation de x
*/
      c->suiv = NULL;
   }
   return (c);
}

int appartient (int x, liste *l)
{
   if (l==NULL)
      return (FAUX);
   else if (l->elt==x)
      return (VRAI);
   else
      return (appartient(x, l-
>suiv));
}


liste *position_element(int x, liste
*l)
{
   if (l==NULL)
      return (NULL);
   else if (l->elt==x)
      return (l);
   else
      return (position_element(x, l-
>suiv));
}


liste *insere_en_tete(int x , liste
*l )
{
liste *c;
   c = (liste *) calloc(1,
sizeof(liste));
   if (c!=NULL)
   {
      c->elt=x;   /* assignation de x
*/;
      c->suiv=l;  /* assignation de
suivant */
      l=c;        /* chainage */
   }
   return (l);
}

liste *insere_liste(int x , liste *l
)
/* verifie si x appartient a l avant
d'insérer */
{
if (! appartient(x, l))
        l=insere_en_tete(x, l);
   return (l);
}


liste *union_liste(liste *a, liste
*b)
{
liste *aux, *c=NULL;
/* recopie des éléments de a dans c
*/
   aux=a;
```

```
   while (aux!=NULL)
   {
      c=insere_en_tete(aux->elt, c);
      aux=aux->suiv;
   }
/* recopie des éléments de b dans c
*/
   aux=b;
   while (aux!=NULL)
   {
      if (! appartient(aux->elt, a))
         c=insere_en_tete(aux->elt,
c);
      aux=aux->suiv;
   }
   return (c);
}

liste *inter_liste(liste *a, liste
*b)
{
liste *aux, *c=NULL;
/* recopie des éléments de b
appartenant à a dans c */
   aux=b;
   while (aux!=NULL)
   {
      if (appartient(aux->elt, a))
         c=insere_en_tete(aux->elt,
c);
      aux=aux->suiv;
   }
   return (c);
}

liste *difference_liste(liste *a,
liste *b)
{
liste *aux, *c=NULL;
/* recopie des éléments de b
n'appartenant pas à a dans c */
   aux=b;
   while (aux!=NULL)
   {
      if (! appartient(aux->elt, a))
         c=insere_en_tete(aux->elt,
c);
      aux=aux->suiv;
   }
   return (c);
}

/* affiche une liste d'entiers ;
retourne le nombre d'éléments */
int affiche_liste(liste *l)
/* version itérative */
{
   int n=0;
   liste *c;
   c=l;
   while (c != NULL)
   {
      printf("%d ",c->elt);
      c=c->suiv;
      n++;
   }
   return (n);
}

/* affiche une liste d'entiers */
int affiche_liste_r(liste *l)
/* version récursive */
{
   if (l != NULL)
   {
      printf("%d ",l->elt);
      affiche_liste_r(l->suiv);
   }
printf("\n");
}

void main (void)
{

liste *a, *b, *c;
int i, j;
int n;

/* premier ensemble */
a=NULL;
printf("Tapez un nombre [1..25]
?\n");
scanf("%d",&n);
srand(n);
for (i=0; i<n; i++)
   a=insere_liste(rand() % 10, a );
j=affiche_liste(a);
printf("\n%d éléments\n",j);


/* deuxiéme ensemble */
b=NULL;
printf("Tapez un nombre [1..25]
?\n");
scanf("%d",&n);
   /* création de la liste b */
for (i=0; i<n; i++)
   b=insere_liste(rand() % 10, b );
j=affiche_liste(b);
printf("\n%d éléments\n",j);


printf("/* union */\n");
c=NULL;
c=union_liste(a, b);
   j=affiche_liste(c);
   printf("\n%d éléments\n",j);


printf("/* intersection */\n");
c=NULL;
c=inter_liste(a, b);
   j=affiche_liste(c);
   printf("\n%d éléments\n",j);


printf("/* différence */\n");
c=NULL;
c=difference_liste(a, b);
   j=affiche_liste(c);
   printf("\n%d éléments\n",j);
}
```

• Pile et file

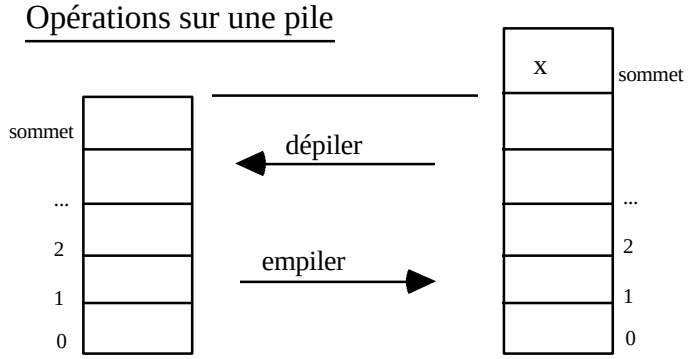
Si l'ordre d'insertion et de suppression des éléments dans la liste importe, deux structures de données sont particulièrement adaptées : la pile et la file.

Pile [stack, lifo]

Une pile est une liste qui respecte la régle "dernier arrivé, premier sorti", alors que la file respecte la régle "premier arrivé, premier sorti".

La **pile** est une structure de données pour laquelle l'ajout et la suppression d'un élément ne sont autorisés qu'à une seule extrémité, appelée *sommet* de la pile. Les opérations sur les listes sont —*P* est une pile, x un élément— :
Vider(P) [*clear()*] *P* ne contient plus aucun élément.
Vide(P) VRAI ssi *P* est vide.
Premier(P) retourne le premier élément de *P*.
Dépiler(P) [*pop()*] retourne le premier élément de *P* et supprime celui-ci de *P*.
Empiler(x, P) [push()] place x dans *P* comme premier élément.

Opérations sur une pile



Il est assez facile de réaliser l'implantation d'une pile à l'aide d'un tableau. Il faut seulement définir une variable auxiliaire, le *pointeur de pile*, qui indique l'adresse du sommet de la pile (qui est aussi le nombre d'éléments de celle-ci).

```
/* Pile de flottants */
#include <stdio.h>
#include <stdlib.h>

#define MAXELEMENT 20
typedef float typepile;

typepile pile[MAXELEMENT];

int pp=0 ;  /* pointeur de pile */

void clear(void)
{
  pp=0;
}

int pile_vide(void)
{
  return (pp==0);
}

int pile_pleine(void)
{
  return (pp==MAXELEMENT);
}

typepile premier_pile(void)
{
  return(pile[pp-1]);
}

typepile pop(void)
/* ne doit pas être utilisé si la
pile est vide */
{
  pp--;
  return(pile[pp]);
}

void push(typepile x)
/* ne doit pas être utilisé si pile
pleine */
{
  pile[pp]=x;
  pp++;
}

/* programme principal */
void main (void)
{
int i, j;
int n;

/* remplir la pile */
printf("Tapez un nombre [1..25]
?\n");
```

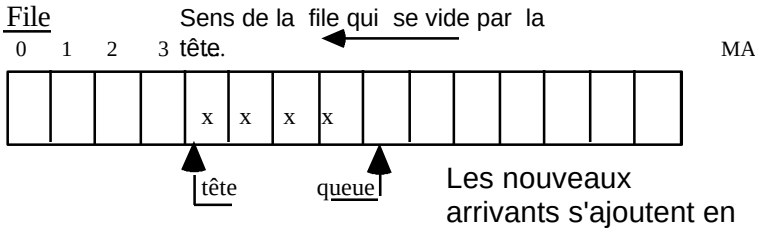
```
scanf("%d",&n);
srand(n);
for (i=0; i<n; i++)
   if (!pile_pleine())
      push(rand() % 10);

printf("\n%d éléments\n",pp);
/* vider la pile */
while(! pile_vide())
   printf("%3.2f ",pop());
printf("\n");


}
```

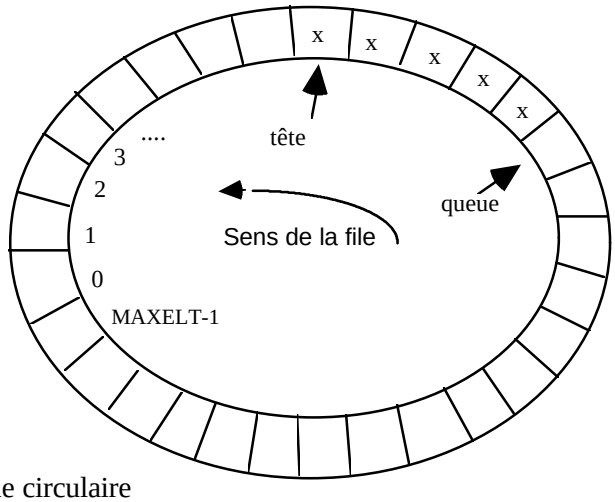
File, queue [queue, fifo]

Une **file** est une structure de données pour laquelle l'ajout et la suppression d'un élément ne sont autorisés qu'aux seules extrémités, appelées la tête et la queue de la file. Les éléments sont ajoutés en queue de file et sont retirés en tête de file. Les opérations sur les listes sont —*F* est une file, x un élément—:

Vider(F) *F* ne contient plus aucun élément.
Vide(F) VRAI ssi *F* est vide.
Premier(F) retourne le premier élément de *F*.
Dernier(F) retourne le dernier élément de *F*.
Défiler(F) retourne le premier élément de *F* et supprime celui-ci de *F*.
Enfiler(x, F) place x dans *F* comme dernier élément.

On dispose de deux index, *tête* et *queue*, et d'un tableau de taille MAXELEMENT, dont les adresses sont désignées modulo MAXELEMENT pour obtenir une structure de données circulaire...





File circulaire

```
/* File circulaire de flottants */
#include <stdio.h>
#include <stdlib.h>

#define MAXELEMENT 20
typedef float typefile;

typefile file[MAXELEMENT];

int tf=0; /* tete de file */
int qf=0;   /* queue de file */

void clear(void)
{
  tf=0;
  qf=0;
}

int file_vide(void)
{
  return (tf==qf);
}

}

int file_pleine(void)
{
  return ((qf+1) % MAXELEMENT == tf);
}

typefile premier_file(void)
{
  return(file[tf]);
}
```

```
typefile dernier_file(void)
{
  if (qf>0)
    return(file[qf-1]);
  else
    return(file[MAXELEMENT-1]);
}

typefile defile(void)
/* ne doit pas être utilisé si la
file est vide */
{
  if (tf<MAXELEMENT-1)
  {
    tf=tf+1;
    return(file[tf-1]);
  }
  else
  {
    tf=0;
    return(file[MAXELEMENT-1]);
  }

}

void enfile(typefile x)
/* ne doit pas être utilisé si file
pleine */
{
  file[qf]=x;
  qf=(qf+1) % MAXELEMENT;
}

/* programme principal */
void main (void)
{
int i, j, n;

/* remplir la file */
printf("Tapez un nombre [1..25]
?\n");
scanf("%d",&n);
srand(n);
j=0;
for (i=0; i<n; i++)
  if (!file_pleine())
  {
    enfile(rand() / 10.0);
    j++;
  }
printf("\n%d éléments\n",j);

while(! file_vide())
  printf("%3.2f ",defile());
printf("\n");
}
```

L'implantation d'une pile ou d'une file sous forme de liste chaînée ne présente pas de difficulté particulière.

• **Hachage**

Vecteur booléen

Soit E un sous-ensemble de l'ensemble d'entiers {0, 1, 2,...,N}, par exemple **E** = {2, 7, 9}
Pour représenter l'ensemble **E**, on utilise une fonction caractéristique *F*:

F :**E** --> {0, 1}
x --> *F*(x)==0 si x n'appartient pas à E
x --> *F*(x)==1 si x appartient à E

Dansnotre exemple E = {2, 7, 9}

0	0	1	0	0	0	0	1	0	1	0	...	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*...*	*N*

Si pour représenter la fonction F on utilise un vecteur de bits, il faut modifier la déclaration précédente en regroupant 8 valeurs consécutive en un seul octet. Ainsi

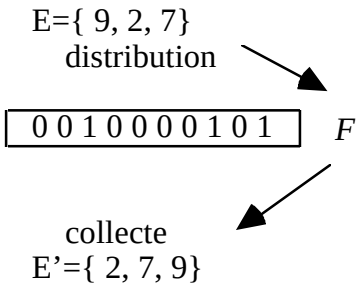
0	0	1	0	0	0	0	1	0	1	0	...	0
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*...*	*N*

peut se représenter par

binaire	0010 0000	1010 0000	0000 0000	....
hexadécimal	20	A0	00	...

Tri par distribution

Le tri par distribution s'aparente au tri postal.
Soit E un sous-ensemble de l'ensemble d'entiers {0, 1, 2,...,N-1}. L'ensemble E a *n* éléments —Card(E)=n— à classer dans l'ordre usuel sur les entiers (ordre croissant). La méthode procéde en deux étapes :
- distribution de E sur un vecteur de bits par une fonction caractéristique *F*
- collecte des éléments non nuls du vecteur de bits parcouru dans l'ordre croissant



La complexité de cet algorithme est proportionnelle au cardinal de E —O(n)—

Hachage

Pour mémoriser des ensembles d'objets (pas nécessairement des entiers) sur lesquels s'appliquent les opérations ensemblistes AJOUTER, SUPPRIMER, ELEMENT, on reprend l'idée du vecteur booléen (ou vecteur de bits). Mais ici la fonction caractéristique, appelée fonction de hachage, est calculée pour un attribut de l'objet, qui peut être numérique, considéré comme une **clé d'accès** à

l'objet. Typiquement, ce sera le cas des objets d'une base de données, dont les tuples sont accessibles par une clé unique.

Fonction de hachage

La fonction de hachage répartit les objets en paquets [*buckets*] —d'où son nom— dont l'adresse est rangée dans une table de hachage. La fonction de hachage établit une relation, pas nécessairement injective, entre la valeur de la clé et son indice dans la table ; plusieurs clés peuvent donc avoir la même valeur de hachage (collision).

On mémorise un sous-ensemble **E** du domaine *D* avec une table **T** de taille *N*.
La fonction *h* associe à chaque clé *x* une valeur de l'intervalle [0.. N-1]

D --> [0,1, ... ,N-1]

x --> h(x)

Le choix de la fonction de hachage doit être considéré avec soin. La taille de la table doit, si possible, être de l'ordre de grandeur de l'ensemble E.Toutes les valeurs entre 0 et N-1 doivent être équiprobables, c'est-à-dire que pour toute clé x et pour tout i entre 0 et N-1, la probabilité que h(x) ait pour valeur i doit être égale à 1/N (h uniforme).

Exemple

Un sous-ensemble **E** du domaine *D* des prénoms

E = {Anne, Guy, Lou, Luc}

D = { x / x est un prénom codé en ASCII }

h(x) = (Somme des codes ASCII de x) MODULO 8

Une table de 8 éléments suffit.

Calcul des valeurs de hachage

h(Anne)= (65+110+110+101) % 8 = 2

h(Guy)= (71+117+121) % 8 = 5

h(Lou)= (76+111+117) % 8 = 0

h(Luc)= (76+117+99) % 8 = 4

T	Lou		Anne		Luc	Guy		
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Si la fonction n'est pas injective (x y et h(x)=h(y)) il se produit des collisions.

h(Paul)= (80+97+117+108) % 8 = 2 = h(Anne)

Exercice : Calculez la valeur de hachage de Noe et placez-le dans la table.

Adressage dispersé

Pour éviter les collisions, on cherche à éparpiller au maximum les données, dans une table qui est de l'ordre de grandeur du domaine.

Dans le cas des prénoms, avec une fonction de hachage

h(x) = (Somme(Ascii(x[i])) % N

quelle taille N donner à la table pour disperser 200 prénoms ? Quel est l'ordre de grandeur de *D* ? — Il y a au moins un prénom par jour du calendrier !—.

Re-hachage

Pour résoudre le problème des collisions, une méthode simple consiste à reporter les clés qui entrent en collision sur les cases libres du tableau. Formellement, cela revient à associer à chaque valeur *x* une suite d'indices dans la table ; le premier indice est h(*x*) ; les indices suivants sont obtenus par un nouveau hachage ou en recherchant la première case libre suivante, circulairement...

((h(x)+k) **modulo** N / k = 0, 1, 2, ...)

T	Lou		Anne	Paul	Luc	Guy	Noe	

0	1	2	3	4	5	6	7

Après ajout de Paul et Noe, les cases *2, 3* et *6* sont occupées par des éléments ayant une valeur de hachage identique.
Pour retrouver Noe, il faut :
- calculer h(Noe) = 2
- comparer T[h(Noe)] et Noe
  si identité SUCCES
  sinon parcourir le tableau circulairement jusqu'à
    - trouver Noe : SUCCES
    - trouver une case libre : ECHEC

Suppression d'un élément

La suppression de l'élément Anne libére une case. Mais alors le fil conducteur jusqu'à Noe est coupé. Donc les cases libérées doivent être marquées VIDE, et les case qui n'ont jamais été occupées marquées BLANC.

T

Lou	BLANC	VIDE	Paul	Luc	Guy	Noe	BLANC
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

Quelques fonctions de hachage

Une fonction de hachage doit être uniforme (la probabilité que h(x)=i doit être proche de 1/N), déterministe (pour une clé donnée elle calcule toujours la même valeur), et facilement calculable.
- Extraction de bits

La clé est toujors représentée par une suite de bits, dont on peut extraire p bits, résultant un nombre entre 0 et 2^p-1. Elle est simple à calculer mais ne donne de bons résultats que si les bits écartés ne sont pas significatifs.
- Compression de bits

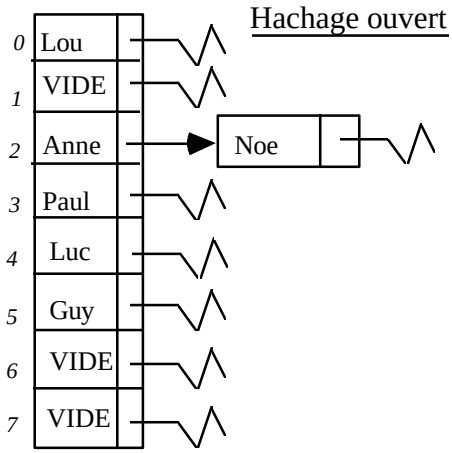
La représentation de la clé est découpée en q tranches de p bits, que l'on combine par une opération telle que l'addition ou le '**ou exclusif'** ;
si x = t[1]t[2]...t[q] alors h(x) = t[1] xor t[2] xor ... xor t[q].
Pour briser certaines répétitions de bits, on peut décaler circulairement les tranches avant de les combiner.
- Division

On prend le reste de la division par N de la représentation de la clé: h(x) = x mod N. Il faut faire attention aux effets indésirables d'accumulation.
- Multiplication

soit un nombre réel r, tel que 0 < r < 1, on calcule
h(x) = Troncature(((x*r) mod 1) * N)
La valeur de r ne doit pas être trop proche de 0 ou de 1 pour éviter des accumulations aux extrémités du tableau. De bonnes valeurs pour r sont (5 - 1) / 2 et 1 - (5 - 1) / 2.

Hachage ouvert

On peut résoudre le problème des collisions en associant une liste chaînée à chaque case de la table de hachage.



• Tri lexicographique

On considère un ensemble de mots ; une table indexée par les lettres de l'alphabet et une fonction de hachage qui à un mot fait correspondre sa i^{éme} lettre.

Donner un algorithme permettant de classer les mots dans l'ordre lexicographique (alphabétique).

Indication : pour classer des mots de même longueur, il suffit de les classer successivement suivant la dernière lettre, puis avant-dernière lettre, etc.

• Dictionnaire d'un texte

On considère un texte T constitué de mots (chaîne ASCII alphanumérique) séparés par des codes <ESPACE>, <TAB>, <NL>, <CR>

Programmer en C le dictionnaire du texte par fonction de hachage fermée.

A chaque mot sera associé un triplet

<mot, position de la première occurrence, nombre d'occurrences>

Les mots seront "case sensitive" (MAJUSCULES et minuscules)

Proposer plusieurs fonctions de hachage et afficher la distribution des valeurs de hachage et le nombre de collisions en fonction de la taille de la table de hachage...

Arbres et ensembles ordonnés

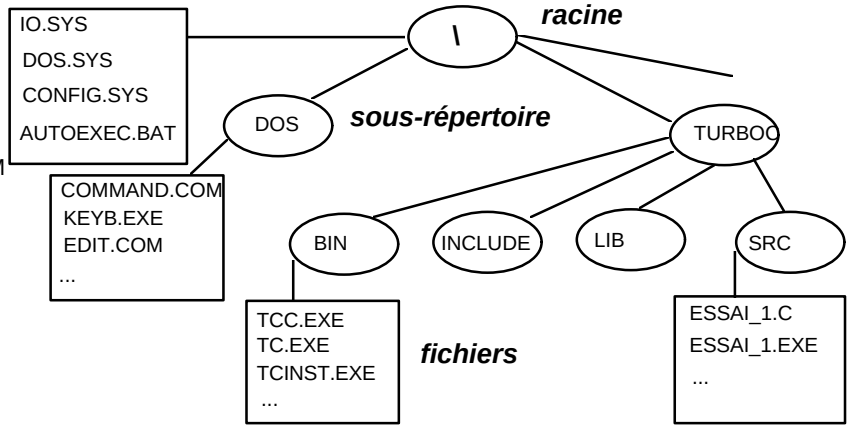
Il est souvent nécessaire d'ordonner les éléments d'un ensemble, par exemple pour améliorer la rapididé d'une recherche... Or le maintien d'un ordre entre les éléments d'un tableau ou d'une liste est relativement coûteux. Pour un tableau par exemple l'insertion d'un élément nécessite de déterminer sa place, en parcourant le tableau depuis le début (*k* comparaisons) puis de décaler les (*n-k*) éléments successeurs pour ménager une place. Donc une complexité en O(*n*) avec *n* le nombre d'éléments du tableau.

Les arbres de recherche sont des structures de données qui permettent de réduire la complexité en temps —mais pas la complexité de la programmation !— des algorithmes d'insertion et de recherche en accédant aux données par dichotomie.

Exemples d'arbres

La structure de fichiers sous DOS —inspirée d'UNIX— organise les fichiers en collections hiérarchisées appelées répertoires [*directory*]. Chaque répertoire (sauf la racine qui n'a pas de pére) a un seul répertoire pére et zéro ou plusieurs sous-répertoires fils. Les fichiers de données proprement dits sont les feuilles de l'arbre.

```
C:\     (Racine)
+--IO.SYS
+--DOS.SYS
+--CONFIG.SYS
+--AUTOEXEC.BAT
+--DOS
    +--COMMAND.COM
    +--KEYB.EXE
    +--EDIT.COM
    +--FORMAT.EXE
    +-- ...
+--TURBOC
    +--BIN
        +--TCC.EXE
        +--TC.EXE
        +--TCINST.EXE
        +--ESSAI_1.EXE
        +-- ...
    +--INCLUDE
        +-- ...
    +--LIB
        +-- ...
    +--SRC
        +--ESSAI_1.C
        +--ESSAI_2.C
        +--ESSAI_1.OBJ
        +--ESSAI_1.EXE
```

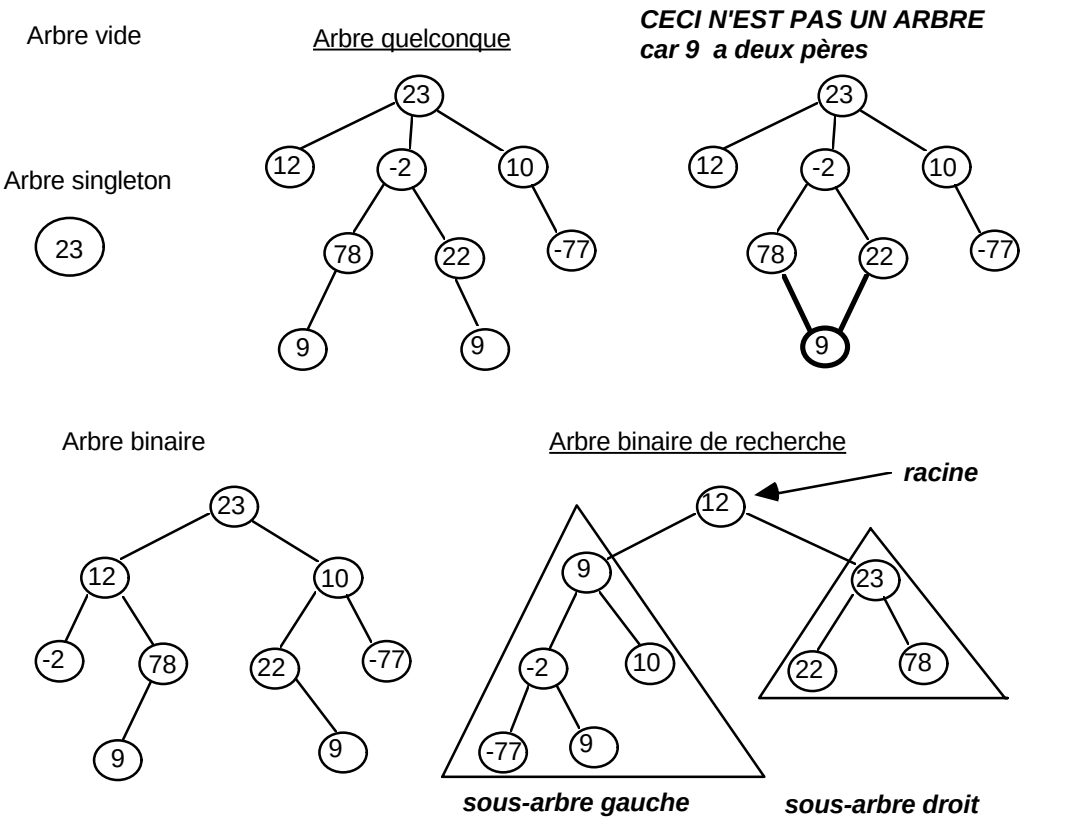


La structure de fichiers et répertoires DOS

L'intérêt de cette organisation est de laisser à l'utilisateur le soin de regrouper les fichiers à sa convenance tout en maintenant une structure hiérarchique.

Un arbre généalogique représentant la relation de parenté "est la mére de" est aussi une structure d'arbre quelconque. Par contre la relation "a pour parents" ne crée pas un arbre (au sens informatique) mais un graphe, un noeud pouvant avoir deux ascendants.

Représentation symbolique des arbres



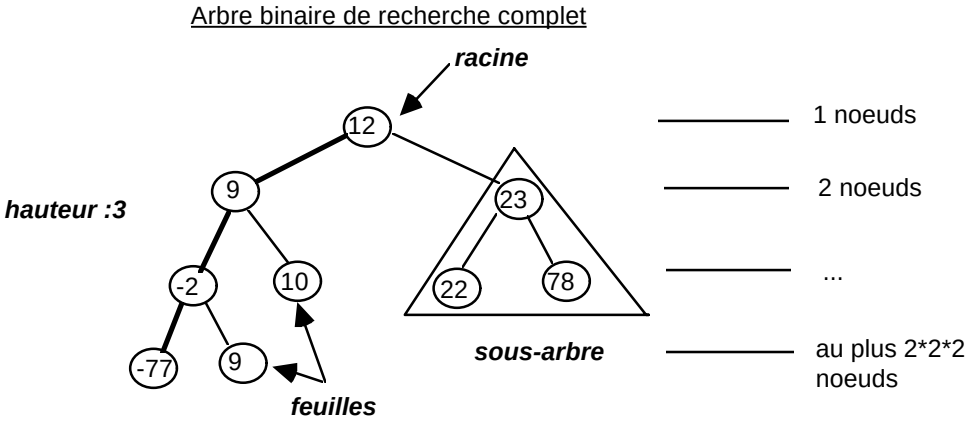
Chaque étiquette d'un noeud du sous-arbre gauche est inférieure ou égale à l'étiquette de la racine qui est inférieure aux étiquettes du sous-arbre droit... cette propriété est maintenue dans l'arbre tout entier.

Par définition un arbre est une structure de données constituée d'un *noeud* appelé racine et de sous-arbres fils de la racine. C'est donc une définition récursive.
Chaque noeud d'un arbre a un ou zéro pére et zéro ou plusieurs fils. Un noeud sans pére est la *racine* de l'arbre. Un noeud sans fils est une *feuille*. Un noeud ayant un pére et au moins un fils est un noeud interne. L'*étiquette* d'un noeud est un élément de l'ensemble représenté par l'arbre. Les liens entre un noeud et ses sous-arbres fils sont les branches de l'arbre.
La *hauteur* de l'arbre est le plus long chemin (sans retour en arriére) entre la racine et une feuille.

Un *arbre vide* n'a aucun noeud. Un singleton a une racine sans fils. Un *arbre binaire* est un arbre dont chaque noeud a au plus deux fils. Un *arbre binaire de recherche* est un arbre binaire dont les étiquettes sont ordonnées selon une *clé* et tel que toutes les clés du sous-arbre gauche sont inférieures ou égales à la clé de la racine, et celles du sous-arbre gauche sont supérieures, et ceci pour tous les sous-arbres...
La hauteur de l'arbre permet d'estimer le nombre *n* d'éléments de celui-ci. En particulier si A est un arbre binaire complet de hauteur h (tous les noeuds ont zéro ou deux fils) il a au plus

$2^0 + 2^1 + 2^2 + .. + 2^{h-1} + 2^h$ éléments

Card(A) $\sum_{i=0,h} 2^i = 2^{h+1} - 1$



Définition axiomatiques des arbres

• Ensembles

Arbre : (N, P), N : ensemble de noeuds, P : relation “pére de”

$Arbre^+$: Arbre - { }

$Arbre_2$: Arbres binaires

Element : Etiquette des noeuds de l’arbre.

L_A : Liste d’arbres.

• Opérations

Arbre-vide : ...--> Arbre : Créer un arbre vide

Racine : Arbre --> Noeud : Retourner le noeud racine d’un arbre

Fils : Arbre --> L_A : Retourner la liste des sous-arbres de la racine

Gauche : $Arbre_2^{+-}$--> $Arbre_2$: Retourner le sous-arbre gauche de la racine

Droit : $Arbre_2^+$--> $Arbre_2$: Retourner le sous-arbre droit de la racine

Cons : Noeud x L_A -->$Arbre^+$: Retourner l’arbre construit à partir du noeud

$Cons_2$:Noeud x $Arbre_2$ x $Arbre_2$ -->$Arbre_2^+$: Retourner l’arbre binaire construit

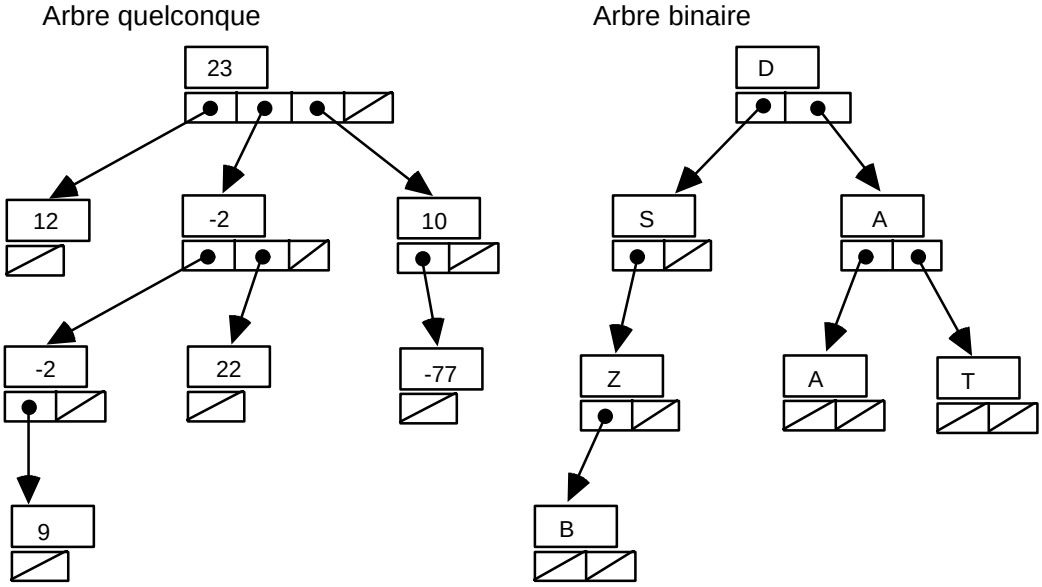
Vide : Arbre --> {VRAI, FAUX} : Tester si un arbre est vide

Elt : Noeud --> Element : Retourner l’étiquette d’un noeud

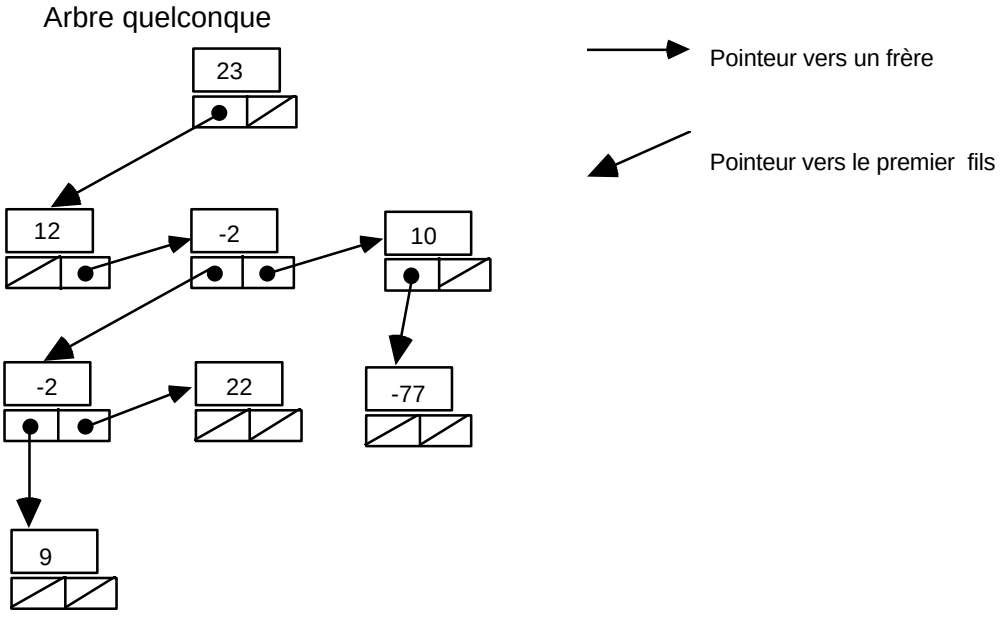
Nouveau : Element --> Noeud : Créer un nouveau noeud

Représentation informatique

La représentation informatique des arbres peut se faire à l'aide de tableaux. On gagne alors en simplicité de programmation ce qu'on perd en souplesse. Mais les arbres étant très bien adaptés à la programmation avec allocation dynamique de mémoire il est préférable d'implanter les arbres sous forme de cellules (record, structure) et de pointeurs.



Pour éviter d'avoir à gérer un nombre variable de pointeurs par noeud d'un arbre quelconque, on peut préférer la structure suivante — fils gauche, frére droit—



Implantation en Langage C

• Il faut d'abord représenter un Noeud.
Par exemple pour les arbres quelconques :

```
typedef struct cellule {
  type_element elt;
  struct cellule * fils_gauche;
```

```
  struct cellule * frere_droit;} noeud;
```

Et pour un arbre binaire :

```
typedef struct cellule {
  type_element elt;
  struct cellule * fils_gauche;
  struct cellule * fils_droit;} noeud;
```

• Un Arbre est un pointeur sur un Noeud, la racine de l'arbre :

```
typedef noeud *arbre;
```

• Un Arbre vide est un pointeur NULL :

```
arbre = NULL;
```

• Opérations

• Vide consiste à tester si la racine est un pointeur NULL :

```
int vide (arbre racine)
{
  return (racine == NULL);
}
```

• Elt : consiste à retourner l'étiquette d'un noeud :
par exemple pour les arbres quelconques :

```
type_element element(arbre racine)
{
  return (racine->elt);
}
```

• Nouveau : il faut faire une allocation mémoire et placer l'étiquette. En cas d'erreur d'allocation le pointeur renvoyé est NULL (l'arbre est vide) :

```
arbre nouveau_binaire(type_element elt, arbre racine)
{
  racine = (struct cellule *) calloc(1, sizeof (struct cellule));
  if (! vide(racine))
  {
    racine->elt = elt;
    racine->fils_gauche= NULL;
    racine->fils_droit=NULL;
  }
  return(racine);
}
```

• Cons : il faut relier un noeud à un ou plusieurs sous arbres.

```
arbre cons_binaire(arbre racine, arbre ss_arbre_g, arbre ss_arbre_d)
{
  racine->fils_gauche = ss_arbre_g;
  racine->fils_droit = ss_arbre_d;
  return(racine);
}
```

• Insertion

Les opérations d'insertion, suppression et recherche d'un élément se programment de façon récursive. Dans le cas d'un arbre binaire de recherche, l'insertion est guidée au cours de la descente dans l'arbre par la relation d'ordre sur les clés.
Il faut donc disposer d'une fonction de comparaison des clés, qui de façon classsique en langage C retourne un entier négatif, nul ou positif selon l'ordre sur les clés :

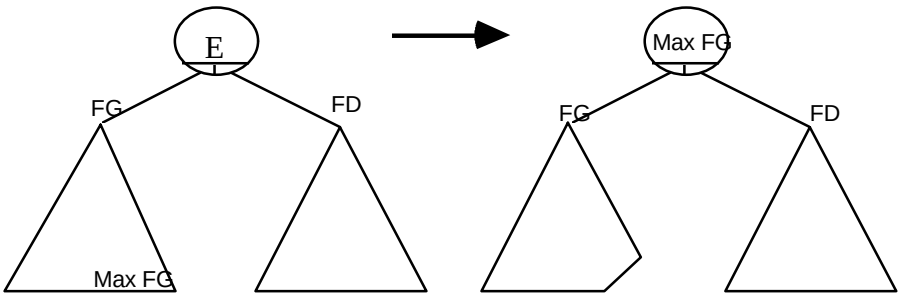
```
int clef(type_element e)
/* renvoie la valeur de la clé */
{
  return (e.cle);
```

```
  }

  int compare(type_element e1, type_element e2)
  {
    if (clef(e1) < clef(e2) ) return (-1);
    else if (clef(e1) == clef(e2) ) return (0);
    else return (1);
  }

  arbre inserer_bin_recherche(type_element elt, arbre racine)
  {
      if (vide(racine))
         racine=nouveau_binaire(elt, racine);
      else
         if (compare(elt, racine->elt) <= 0)
           racine_fils_gauche = inserer_bin_recherche(elt, racine-
>fils_gauche);
         else
           racine->fils_droit = inserer_bin_recherche(elt, racine->fils_droit);
      return(racine);
  }
```

La complexité de cet algorithme est proportionnelle à la hauteur de l'arbre et donc en $O(\log_2 n)$ pour un arbre binaire de recherche équilibré.

- **Suppression**

La suppression d'un élément doit conserver l'arbre binaire de recherche. Il faut donc remplacer l'étiquette supprimée par l'étiquette la plus à droite du sous-arbre gauche.



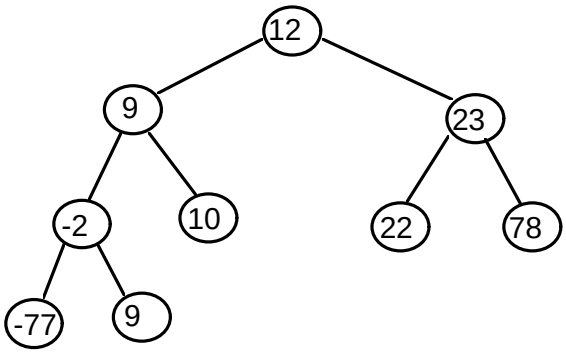
La suppression de l'étiquette E nécessite de remplacer celle-ci par la plus grande étiquette du sous-arbre gauche...

La programmation de cette opération assez délicate sera laissée à la sagacité du lecteur !

- **Parcours d'arbre**

Un parcours en profondeur consiste à descendre dans l'arbre depuis la racine en commençant par le premier fils (le fils gauche pour un arbre binaire), de façon récursive, puis à traiter les autres fils. Selon l'ordre d'affichage de l'étiquette des noeuds —avant, entre ou après les fils— l'affichage sera préfixe, infixe ou suffixe.

Parcours en profondeur d'un arbre binaire de recherche



Parcour s préfixe : 12, 9, -2, -77, 9, 10, 23, 22, 78

Parcours infixe : -77, -2, 9, 9, 10, 12, 22, 23, 78

Parcours suffixe : -77,9, -2, 10, 9, 22, 78, 23, 12

Le parcours infixe affiche les éléments dans l'ordre croissant.

```
void infixe(arbre racine)
{
    if (! vide(racine))
    {
       infixe(racine->fils_gauche);
       printf("%d\t",racine->elt);
       infixe(racine->fils_droit);
}
```

Exercices

A est un arbre binaire de recherche dont les données sont des entiers.

1) Donner en langage C la définition de A.

2) Dessiner A après insertion des nombres 100, 20, 30, 150, 110, 10, 25, 45, 150, 200.

3) Quelle est la hauteur de l'arbre A ?

4) Que devient l'arbre A si

- on supprime 30 ?
- on supprime 100 ?
- on rajoute 5 et 120 ?

5) Ecrire une fonction affichant les éléments d'un arbre binaire de recherche dans l'odre décroisssant.

6) Le parcours en largeur consiste à afficher la valeur de chaque noeud au fur et à mesure de la descente dans l'arbre, puis à traiter chaque fils dans l'ordre gauche puis droite. Pour l'arbre de la question 2 un parcours en largeur produit la liste :

100, 20, 10, 30, 25, 45, 150, 110, 200

Ecrire en C la fonction

```
void parcours_largeur(type_arbre *racine);
```

7) Ecrire une fonction "taille" retournant le nombre de noeuds d'un arbre.

8) Ecrire une fonction "hauteur" retournant la hauteur d'un arbre.

```
/* JF */
/* Arbre binaire de recherche */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define MAXCAR 20
#define MAXHACHE 1000;

typedef struct cellule {
     char elt[MAXCAR];
     struct cellule * fg;
     struct cellule * fd;
     } *arbre;

#define MAXHAUTEUR 100

struct liste {char n[MAXCAR]; struct
liste * suiv;} *ta[MAXHAUTEUR];  /*
tableau de listes */

FILE *f;

/* PROTOTYPES */

arbre insere(char *element, arbre
racine);  /* insere un element dans
l'arbre binaire */
arbre recherche (char *element, arbre
racine);  /* recherche un element */
arbre min_arbre(char *element, arbre
racine);  /* supprime le min de
l'arbre */
arbre supprime(char *element, arbre
racine);  /* supprime element de
l'arbre */
void affiche_p(arbre racine);      /*
affiche en profondeur infixe */

struct liste *
insere_queue_liste(struct liste * l,
char n[]);  /* ajoute eun element n
queue de liste */
void cree_l(int h, arbre racine);
       /* transforme un arbre en
tableau de listes */
void affiche_liste(struct liste *l);
         /* affiche le tableau de
listes niveau par niveau */
void affiche_l(arbre racine);
     /* affiche un arbre en largeur
*/

/* FONCTIONS */

arbre insere(char *element, arbre
racine)
/* insere un element dans un arbre */
{
  if (racine==NULL) /* creer l'arbre
*/
  {
    racine = (struct cellule *)
calloc(1, sizeof(struct cellule));
    if (racine != NULL)
    {
      strcpy(racine->elt, element);
      racine->fg = NULL;
      racine->fd = NULL;
    }
    else
      perror("Erreur allocation
memoire\n");
  }
  else if (strcmp(element, racine-
>elt)<=0)
      racine->fg=insere(element,
racine->fg);
  else
    racine->fd=insere(element,
racine->fd);
  return (racine);
}

arbre recherche (char *element, arbre
racine)
{
  int test;
  if (racine!=NULL)
  {
    test = strcmp(racine->elt,
element);
    if (test==0)
      return (racine);
  else if (test<0)
      return( recherche(element,
racine->fd));
    else
      return( recherche(element,
racine->fg));
  }
  else return (NULL);
}

arbre min_arbre(char *element, arbre
racine)
/* supprime la valeur minimum de
l'arbre et la place dans element */
{
arbre aux;
  if (racine!=NULL)
  {
    if (racine->fg==NULL)
    {
      aux=racine;
      strcpy(element, racine->elt);
      racine=racine->fd;
      free(aux);
    }
    else
    {
      racine->fg=min_arbre(element,
racine->fg);
    }
  }
  return(racine);
}

arbre supprime(char *element, arbre
racine)
/* supprime element de l'arbre */
{
  int test;
  arbre aux;
```

```
   aux = racine;
   if (racine!=NULL)
   {
     test = strcmp(racine->elt,
element);
     if (test==0)
     {
        /* remplacer cet element */
        if (racine->fd==NULL)
        /* faire monter le sous-arbre
gauche */
        {
          aux=racine;
          racine=racine->fg;
          free(aux);
        }
     else if (racine->fg==NULL)
     /* faire monter le sous-arbre
droit */
     {
        aux=racine;
        racine=racine->fd;
        free(aux);
     }
     else /* remplacer par min_fd */
        {
          racine->fd =
min_arbre(racine->elt, racine->fd);
        }
     }
     else /* rechercher sur les fils
*/
     if (test<0)
        racine->fd=supprime(element,
racine->fd);
     else
        racine->fg=supprime(element,
racine->fg);
   }
  return (racine);
}

void affiche_p(arbre racine)
{
  if (racine!=NULL)
  {
       affiche_p(racine->fg);
       printf("%s\t",racine->elt);
       affiche_p(racine->fd);
  }
}


/* affichage en largeur */

struct liste *
insere_queue_liste(struct liste * l,
char n[])
{
  struct liste *aux;
  if (l==NULL) /* creer la liste */
  {
     l=(struct liste *) calloc(1,
sizeof(struct liste));
     if (l==NULL)
        perror("Erreur allocation
memoire \n");
     else
     {
        strcpy(l->n, n);
        l->suiv=NULL;
/*      printf("Insertion %s\t",n); */
     }
  }
  else
  {
     aux=l;
     while (aux->suiv != NULL)
        aux=aux->suiv;
     aux->suiv=(struct liste *)
calloc(1, sizeof(struct liste));
     if (aux->suiv==NULL)
        perror("Erreur allocation
memoire \n");
     else
     {
        strcpy(aux->suiv->n, n);
        aux->suiv->suiv=NULL;
/*      printf("Adjonction %s\t",n);
*/
     }
  }
  return (l);
}

void cree_l(int h, arbre racine)
{
  if (racine!=NULL)
  {

  ta[h]=insere_queue_liste(ta[h],raci
ne->elt);
       cree_l(h+1,racine->fg);
       cree_l(h+1,racine->fd);
  }
}

void affiche_liste(struct liste *l)
{
     while (l != NULL)
     {
        printf("%s\t", l->n);
         l= l->suiv;
     }
}

void libere_liste(struct liste *l)
{
struct liste *aux;
  while (l != NULL)
  {
     aux=l->suiv;
     free(l);
     l = aux;
  }
}

void libere_l(void)
{
int i ;
for (i=0; i<MAXHAUTEUR; i++)
{
  libere_liste(ta[i]);
  ta[i]=NULL;
```

```
}
}

void affiche_l(arbre racine)
{
int i ;
libere_l();
cree_l(0,racine);
printf("\n");
for (i=0; i<MAXHAUTEUR; i++)
  if (ta[i]!=NULL)
  {
     printf("%d\t",i);
     affiche_liste(ta[i]);
     printf("\n");
  }
}

/* MAIN */


void main (int argc, char *argv[])
{
int i, c;
char str[20];
struct cellule *racine;
i=0;


if (argc>1)
{
  if ((f=fopen(argv[1], "r"))!=NULL)
  {
     printf("Ouverture en lecture de
%s\n",argv[1]);
     while ((c=getc(f))!=EOF)
     {
       if ((c==' ') || (c=='\n') ||
(c=='\t'))
       {
          str[i]=0;
          i=0;
          printf("%s\n",str);
          racine=insere(str, racine);
       }
     else
          str[i++]=c;
     }
     fclose(f);
  }
}

do
{
 printf("MENU : i:inserer s:supprimer
a:arbre l:liste q:quitte\n");
 do {
  c=getchar();
 } while ((c=='\n') || (c=='\t') ||
(c==' '));
/* boucler sans rien faire */

  switch (c) {
  case 'i':
  case 'I' :
         rewind(stdin);
         str[0]=0;
         printf("Etiquette a inserer
?\n");
         if (scanf("%s",str)!=0)
         {
           printf("Insertion  de %s\n
",str);
            racine= insere(str,
racine);
        }
         break;
  case 's':
  case 'S' :
         rewind(stdin);
         str[0]=0;
         printf("Etiquette a
supprimer ?\n");
         if (scanf("%s",str)!=0)
         {
           printf("Suppression de
%s\n ",str);
            racine= supprime(str,
racine);
        }
         break;
  case 'a' :
  case 'A' :
         printf("Affichage en
profondeur \n");
         affiche_p(racine);
         printf("\n");
         fflush(stdout);
         break;
  case 'l' :
  case 'L' :
         printf("Affichage en largeur
\n");
         printf("Niveau\tValeurs
\n");
         affiche_l(racine);
         fflush(stdout);
         break;
  default : break;
  }
} while ((c!='q') && (c!='Q') );

}
```

Graphes

Les graphes interviennent chaque fois qu'on veut représenter et étudier un ensemble de liaisons (orientées ou non) entre les éléments d'un ensemble fini d'objets.

Par exemple représenter un réseau routier, électrique, un circuit électronique, un réseau informatique, l'ordonnancement de tâches, une application hyper-média, etc.

Après des définitions et notions fondamentales sur les graphes, nous présenterons quelques algorithmes élémentaires.

Définition

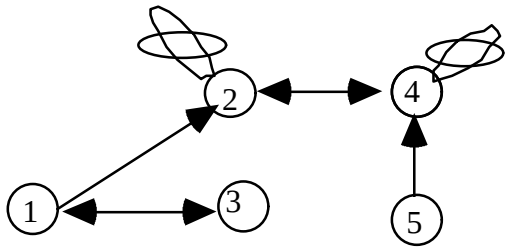
Un graphe G est un couple (S,A) où :
- S est un ensemble **fini** de sommets ;
- A est un sous-ensemble de SxS, ensemble des arcs de G.

On prendra généralement pour S un segment [1,n].

Exemple

Représentation du graphe *orienté*

G_1 = (S1,A1) = ({1,2,3,4,5}, {(1,2), (1,3), (2,2), (2,4), (3,1), (4,2), (4,4), (5,4)})



Un arc (x,y) représente une liaison orientée entre l'origine x et l'extrémité y. Si (x,y) est un arc, x est le prédécesseur de y et y est le successeur de x ; si x=y l'arc est une boucle.

Graphe non orienté

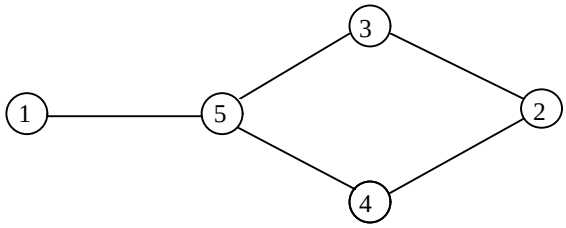
On dit qu'un graphe G = (S,A) est ***non orienté*** si et seulement si, pour tout si et sj de l'ensemble des sommets S, si (si, sj) est un arc de A, alors (sj, si) aussi. Les arcs s'appellent alors des arêtes et sont représentées par des paires {si, sj}, non orientées bien sûr.

Exemple

Le graphe non orienté G_2 = (S2, A2), représenté graphiquement par :

S2 = {1,2,3,4,5};

A2 = {(1,5), (2,3), (2,4), (3,2), (3,5), (4,2), (4,5), (5,1), (5,3), (5,4)}



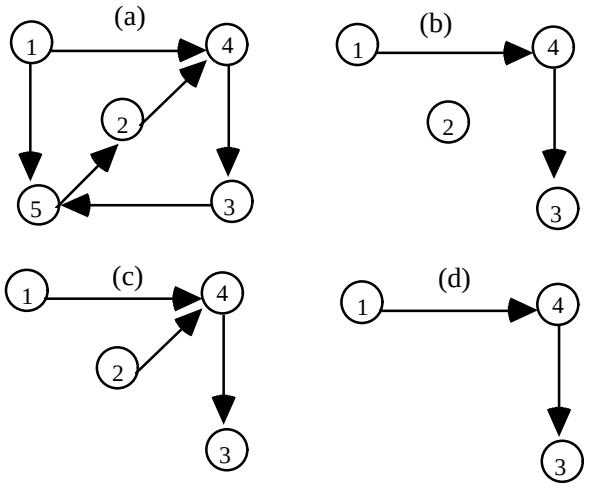
Propriétés

Deux sommets x et y d'un graphe orienté (respectivement non orienté) sont *adjacents* s'ils sont les extrémités d'un arc (respectivement d'une arête).

Soit un graphe G=(S,A) et T un sous-ensemble de S. L'ensemble des arcs de A dont une extrémité est dans T et l'autre est dans S-T est appelé le *cocycle* associé à T. L'ensemble des sommets de S-T adjacents à au moins un sommet de T est la *bordure* de T.
Le graphe G=(S,A) est dit *biparti* s'il existe un sous-ensemble de sommets T tel que A= cocycle(T).

Un sous-graphe du graphe G = (S,A) est un couple G' = (S',A') pour lequel S' est inclus dans S, et A' inclus dans A. Le sous-graphe G' est un graphe partiel de G si S'=S. Si A' est l'ensemble des arcs de A dont les deux extrémités sont dans S', le sous-graphe G' est dit induit par S'.

Exemple :
G3 = (S3, A3) = ({1,2,3,4,5}, {(1,4), (1,5), (2,4), (3,5), (4,3), (5,2)})



(a) Graphe orienté G (b) Un sous-graphe de G
(c) Le graphe partiel induit par les sommets {1, 2, 3, 4}
(d) Le sous-graphe induit par les arcs {(1,4) , (4,3)}
Le graphe G est biparti parce que tout arc est incident à T = {1, 2, 3} (i.e. a l'une de ses extrémités dans T)

Implémentation d'un graphe

Trois représentations reviennent principalement : la matrice d'adjacence, la matrice d'incidence et la liste des successeurs.

Matrice d'adjacence

On associe à un graphe G = ([1,n], A) une matrice carrée d'ordre n à valeurs dans {0,1}, appelée matrice d'adjacence, telle que quels que soient les sommets i, j de l'ensemble des sommets [1,n], la valeur de l'élément $m_{i,j}$ de la matrice est égal à 1 si et seulement si (i,j) est un arc de G, et 0 sinon.

Exemple :
G_1 = ({1,2,3,4,5}, {(1,2), (1,3), (2,2), (2,4), (3,1), (4,2), (4,4), (5,4)}**)**

M1	*1*	*2*	*3*	*4*	*5*
1	0	1	1	0	0
2	0	1	0	1	0
3	1	0	0	0	0
4	0	1	0	1	1
5	0	0	0	1	0

Matrice d'adjacence M1 du graphe orienté G_1.

G2 = ({1,2,3,4,5}, {(1,5), (2,3), (2,4), (3,2), (3,5), (4,2), (4,5), (5,1), (5,3), (5,4)})

M2	*1*	*2*	*3*	*4*	*5*
1	0	0	0	0	1
2	0	0	1	1	0
3	0	1	0	0	1
4	0	1	0	0	1
5	1	0	1	1	0

Matrice d'adjacence M2 du graphe non orienté G_2 (la matrice M2 est symétrique).

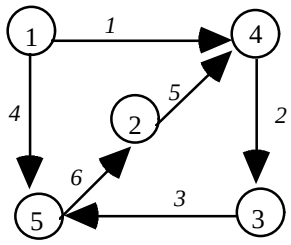
Matrice d'incidence

Si G est un graphe sans boucle, sa matrice d'incidence "sommets-arcs" (G) est une matrice |S|x|A| à valeurs dans {-1,0,1}, telle que l'élément $d_{x,a}$ de la matrice est égal à 1 si x est *l'origine* de l'arc *a*, -1 si x est *l'extrémité* de l'arc *a* et 0 sinon...

Exemple : matrices d'adjacence et d'incidence de G3.
G3 = (S3, A3) avec S3= {**1,2,3,4,5**} liste des sommets
et A3= {*1,2,3,4,5,6* } liste des arcs numérotés *1*=(1,4); *2*=(4,3) ; *3*=(3,5) ; *4*=(1,5) ; *5*=(2,4) ; *6*=(5,2)

M3	**1**	**2**	**3**	**4**	**5**
1	0	0	0	1	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	1	0	0	0

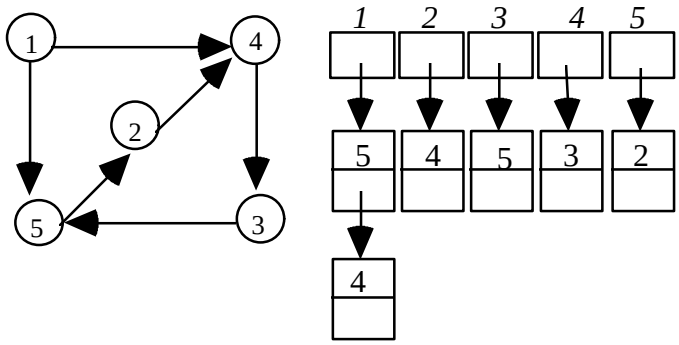
3	*1*	*2*	*3*	*4*	*5*	*6*
1	1	0	0	1	0	0
2	0	0	0	0	1	-1
3	0	-1	1	0	0	0
4	-1	1	0	0	-1	0
5	0	0	-1	-1	0	1



Liste des successeurs ou liste d'adjacence

On peut aussi décrire un graphe par un tableau ($q_1, q_2, ... q_n$) où l'élément q_i est un pointeur sur la liste des successeurs du sommet *i*.

Exemple :



Liste des successeurs pour le graphe G3

Chemins, chaînes, circuits, cycles

Soit G un graphe orienté.

Un *chemin* d'origine x et d'extrémité y est une suite finie non vide de sommets c = ($s_0, s_1, s_2 .,.., s_p$) telle que s_0,=x et s_p=y et pour k= 0, 1, .., p-1, (s_k , s_{k+1}) est un arc du graphe. La *longueur* du chemin est *p* ; c'est le nombre d'arcs (non nécessairement distincts) empruntés par ce chemin. Un chemin est *simple* si les arcs sont deux à deux distincts et il est *élémentaire* si ce sont les sommets qui sont deux à deux distincts... Si *p* est 1 et que $s_0 = s_p$ le chemin est un *circuit* (il revient au départ).

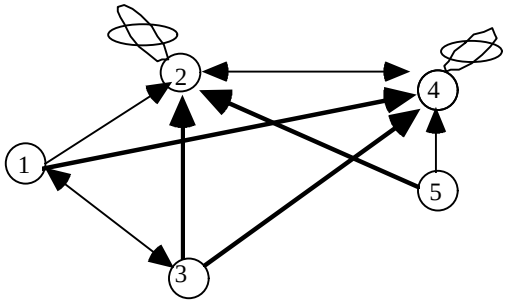
Dans le cas de graphes non orientés, l'équivalent d'un chemin est une *chaîne* et celui d'un circuit est un *cycle*. Un graphe non orienté est connexe si pour tout couple de sommets il existe une chaîne ayant ces deux sommets pour extrémités. Autrement dit on peut aller de tout sommet du graphe à tout autre... Par extension un graphe orienté est connexe si sa version non orientée (obtenue en supprimant les orientations et les boucles) est connexe. La connexité est une relation d'équivalence entre les sommets du graphe et ses classes d'équivalences sont appelées composantes connexes du graphe.

Fermeture transitive d'un graphe

La fermeture transitive d'un graphe G est le graphe G' où deux sommets s_i et s_j sont reliés par un arc si et seulement si il existe un chemin dans G allant de s_i à s_j .

Algorithme de fermeture transitive

Les méthodes de test d'existence d'un chemin entre deux sommets sont fondées sur le lemme de König qui assure que s'il existe un chemin entre deux sommets d'un graphe, il en existe alors un de longueur inférieure ou égale à N, nombre de sommets du graphe. Autrement dit il existe un chemin élémentaire...



Graphe G'1, fermeture transitive du graphe orienté G1

Arcs initiaux
Arc créés par transitivité

Pour calculer la fermeture transitive d'un graphe, nous allons définir d'abord deux opérations sur les matrices d'adjacence à valeurs booléennes :
Soient deux matrices carrées $M=(m_{i,j})$ et $N=(n_{i,j})$ d'ordre N à valeurs booléennes {0,1} ; on définit leur somme S et leur produit P par les formules :

$s_{i,j} = MIN (1, m_{i,j} + n_{i,j})$

$p_{i,j} = MIN (1, (m_{i,k} \times n_{k,j})_{1 \le k \le n})$

Pour calculer la fermeture transitive d'un graphe G de N sommets associé à une matrice M, on définit une suite de matrices d'ordre N, à valeurs dans {0,1} définie par :

$M_1 = M$

$M_p = M \times M_{p-1}$

On montre qu'il existe un chemin de longueur k entre deux sommets i et j de G si et seulement si on a $M_k(i,j)=1$.

Cette remarque combinée avec le lemme de König fournit un algorithme : la somme booléenne des n matrices M_1, M_2 ... M_n n'est autre que la matrice associée à la fermeture transitive de G.

Exercice 1

Ecrire un programme qui posséde les fonctionnalités suivantes :
- Saisie de la matrice d'adjacence associée à un graphe G.
- Affichage de la matrice de la fermeture transitive de G.
- Réponse à la question : “Existe-t'il un chemin de longueur k entre les sommets i et j ?”.

Exercice 2

1) Saisie d'un graphe
2) Affichage d'un graphe
3) Coloriage

1) Saisie.
Un graphe orienté G est un ensemble de sommets S et un ensemble d'arcs A.
Les sommets sont donnés sous forme d'une liste de l'intervalle [1..n], n nombre de sommets. Les arcs sous forme d'une liste de couples (a,b), avec a sommet origine et b sommet extrémité de l'arête.
Exemple
G = (S,A) avec S=[1..6] A={(1,2), (2,5), (3,2), (3,6), (4,3), (5,6)}
Les principales opérations sur les graphes orientés sont la lecture des étiquettes associees aux sommets ou aux arcs, l'insertion ou la suppression de sommets et d'arcs, et le parcours du graphe le long de leurs arcs en suivant l'orientation de ceux-ci.

Question 1
Représenter graphiquement le graphe G1 = (S1,A1)
S1=[1..8]
A1={(1,2), (2,1), (2,4), (3,3), (3,5), (4,4), (5,3), (5,5), (6,1),
(6,2), (6,4), (6,7)}

Plusieurs méthodes permettent de représenter un graphe :
a) Liste (ou table) de sommets et d'arcs (comme ci-dessus)
b) Matrice d'adjacence
c) Liste d'adjacence

Question 2

Représentez les graphes G et G1 par une matrice d'ajacence et par une liste d'adjacence.

Question 3
Définir le type type_sommet du sommet d'un graphe orienté.
Définir le type type_indice de la position d'un sommet dans la liste des sommets adjacents a un sommet du graphe.

Question 4
Ecrire les fonctions C de saisie d'un graphe d'au plus MAXSOMMET et MAXARETE représenté par
a) Tableau d'arcs
```
type_sommet sommets_graphe[MAXSOMMET];
type_sommet aretes_graphe[MAXARETE][2]
```
b) Matrice d'ajacence
```
type_sommet
mat_adj[MAXSOMMET][MAXSOMMET];
```
c) [FACULATIF] liste d'adjacence
```
typedef struct arete{type_sommet a; type_sommet b;
        struct arete *suivant;}arete;
 arete *liste_adj[MAXSOMMET];
```

On prendra soin de définir les fonctions de base de manipulation des graphes pour chacune des implantations :
```
type_indice premier(type_sommet s);
```
retourne l'indice du premier sommet rencontre dans la liste des sommets
adjacents à s (son indice est 0 dans cette liste si on utilise une liste d'ajacence).
```
type_indice suivant(type_sommet s, type_indice i);
```
retourne l'indice du sommet adjacent à s qui suit le sommet d'indice i dans la liste des sommets adjacents à s.
```
type_sommet sommet(type_sommet s, type_indice i);
```
retourne le sommet d'indice i dans la liste des sommets adjacents à s.

2) Affichage.
L'affichage du graphe est destiné à vérifier la qualité de la saisie. Le programme devra aussi afficher les sommets isolés...
Question 5
Ecrire les fonctions C d'affichage de la liste des arcs d'un graphe d'au plus MAXSOMMET et MAXARETE representé par
a) Tableau d'arcs
b) Matrice d'ajacence
c) Liste d'adjacence

3) Coloriage.
Colorier un graphe non orienté consiste à attribuer une couleur à chaque sommet de telle sorte que deux sommets adjacents n'aient pas la même couleur... Un coloriage optimal est celui qui nécessite le moins de couleurs distinctes...
Exemple : le graphe G nécessite 2 couleurs.

Question 6
Combien de couleurs nécessite le coloriage de G1 ?
Question 7
Ecrire les fonctions C de coloriage d'un graphe d'au plus MAXSOMMET et MAXARETE representé par
a) Tableau d'arêtes
b) Matrice d'ajacence
c) Liste d'adjacence
Ces fonctions devront fournir pour chaque sommet le numéro de couleur attribué et le nombre de couleurs utilisées.
Question 8
Proposer une heuristique pour rechercher un coloriage optimal...
Données
Le graphe des carrefours proposé en cours fera l'affaire...

Exercice 3

1) Saisie d'un graphe
2) Parcours en profondeur
3) Parcours en largeur

Un graphe orienté G est un ensemble de sommets S et un ensemble d'arcs A.
Les sommets sont donnés sous forme d'une liste de l'intervalle [1..n], n nombre de sommets. Les arcs sous forme d'une liste de couples (a,b), avec a sommet origine et b sommet extrémité de l'arête.

Représentez graphiquement le graphe G2
G2 = (S2,A2)
S2=[1..7]
A2={(1,2), (2,1), (2,4), (3,3), (3,5), (4,3), (4,6), (5,3), (5,5), (6,1), (6,2), (6,7)}

1) Implantation d'un graphe

Question 1
Définir le type *type_sommet* du sommet d'un graphe orienté.
Définir le type t*ype_indice* de la position d'un sommet dans la listedes sommets adjacents à un sommet du graphe.
Définir les fonctions de base de manipulation des graphes pour la matrice d'adjacence et la liste d'adjacence...

```
type_indice premier(type_sommet s);
```
retourne l'indice du premier sommet rencontré dans la liste des sommets
adjacents à s (son indice est 0 dans cette liste si on utilise une liste d'ajacence).
```
type_indice suivant(type_sommet s, type_indice i);
```
retourne l'indice du sommet adjacent à s qui suit le sommet
d'indice i dans la liste des sommets adjacents à s.

type_sommet sommet(type_sommet s, type_indice i);
retourne le sommet d'indice i dans la liste des sommets adjacents à s.

2) Parcours en profondeur

Question 2
Donner la liste des sommets visités dans un parcours en profondeur d'abord en visitant les sommets de 1 à 7.

Question 3
Ecrire la fonction de parcours en profondeur en utilisant les fonctions définies au 1 pour une représentation par matrice d'adjacence et pour une représentation par liste d'adjacence. Cette fonction affichera les sommets visités dans l'ordre préfixe et les arcs.

Question 4
Ecrire la fonction de parcours en largeur d'abord pour les deux modes de représentation.

Question 5
Ecrire une fonction détectant les circuits dans un graphe.

Tris

La recherche d'un élément dans un ensemble est bien améliorée si les éléments sont ordonnés. C'est en particulier le cas des recherches de mots dans un dictionnaire. Ordonner les élément consiste à effectuer une permutation entre les éléments de sorte qu'après celle-ci les éléments soient classés —triés— en croissant (respectivement décroissant)...

Données :
N, nombre fini des éléments.
Les éléments sont tous du même type et occupent un espace mémoire fini.
Les données sont par exemple représentées dans un tableau.

a[0], a[1], ..., a[N-1] est la liste initiale
b[0], b[1], ..., b[N-1] est la liste finale telle que b[0]< b[1] <... < b[N-1].
P est une permutation qui ordonne les éléments :
b[i] = a[P(i)] avec P une permutation de [0, N[.

Pour effectuer le réarrangement, il faut disposer d'une relation d'ordre total sur les éléments.

Relation d'ordre

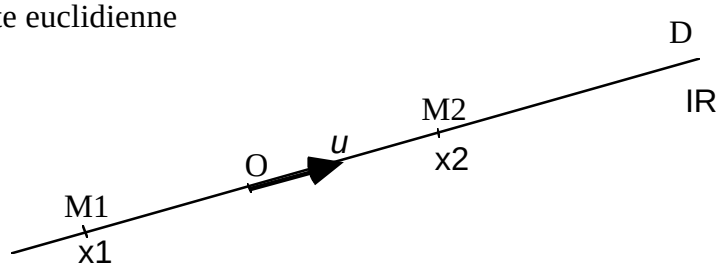
Une relation d'ordre notée *R* sur les éléments d'un ensemble E est une relation binaire ayant les propriétés :
Réflexivité : *aRa*
Transitivité : si *aRb* et si *bRc* alors *aRc*
Antisymétrie : si *aRb* et si *bRa* alors *a=b*.
En général une relation d'ordre est notée <. Elle est d'ordre total si tous les éléments de l'ensemble sont comparables deux à deux.

Pour les données complexes, la relation d'ordre sera définie sur une clé, c'est-à-dire sur une liste d'attributs qui identifient de façon unique chaque élément de l'ensemble.

Exemples de relations d'ordre

Droite réelle D.
On peut définir un ordre sur la droite euclidienne en choisissant un point O appelé origine et un vecteur unitaire *u* lié à l'origine. Cela définit sans ambigüité un sens de parcours.
A chaque point M on associe son abscisse réelle x, c'est-à-dire la mesure algébrique du vecteur *OM* dans le repére (O,*u*). La relation d'ordre sur IR induit une relation d'ordre entre les points de la droite.
M1 "avant ou confondu avec " M2 ssi x1 x2.

La droite euclidienne

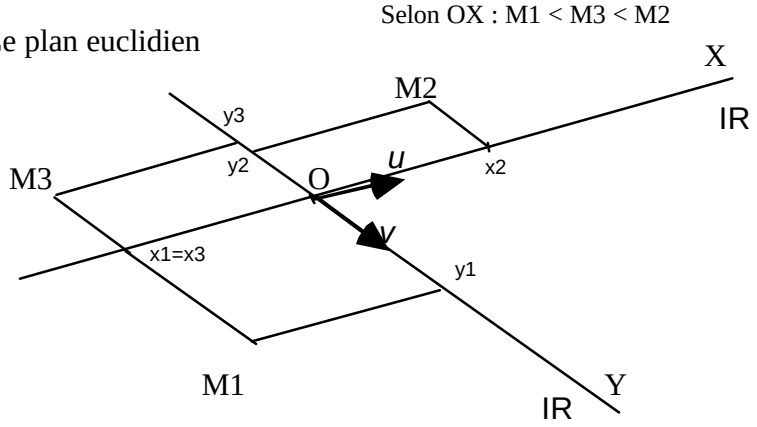


Plan euclidien

On peut définir une relation d'ordre entre les points du plan à deux dimension P de plusieurs façons. Par exemple en utilisant des coordonnées polaires.
Avec un système de coordonnées cartésiennes, M[x, y], par rapport à un repére (O, *u*, *v*) avec *u* et *v* unitaires non colinéaires, il n'est pas évident qu'on puisse déterminer une relation d'ordre unique.

La figure montre que la relation d'ordre des projections sur une seule droite —par exemple OX (droite orientée par *u*)— ne suffit pas à ordonner sans ambiguité les points M1, M2 et M3.
Par contre en combinant la relation d'ordre sur les projections sur OX avec une relation d'ordre sur les projections sur OY, on peut induire un ordre total sur P.

Le plan euclidien



Pour ordonner le Plan euclidien il faut définir un ordre total : par exemple
ordre selon OX, et si les abscisses sont égales, ordre selon OY.
En ce cas M3 < M1 < M2

Fiche de recensement

Un recensement porte sur une population. Chaque élément est représenté par une fiche : (Numéro, Nom, Prénom, Date de naissance, Adresse, Sexe).

Plusieurs ordres sont possibles, selon le numéro, selon le nom et le prénom, etc.
Il importe de définir précisément quelle sera la clé puis on induira l'ordre sur les objets à partir de l'ordre sur les clés.

Opérations élémentaires pour le tri.

Toute opération de tri implique de disposer d'une fonction de comparaison entre deux éléments (ordre des clés), et d'une fonction d'échange (permutation) entre éléments.

Algorithmes de tri

Les algorithmes de tri ont fait l'objet de nombreuses recherches. Ils différent principalement par :
- la simplicité de mise en oeuvre,
- l'efficacité.

En général ce sont deux contraintes contradictoires. Plus un algorithme est efficace, plus il est sophistiqué et délicat à implanter.
Nous commencerons par présenter des algorithmes dits "naïfs" car très immédiats.

Tri par sélection

Soit une liste L quelconque, et une liste vide T.
L= {-5, 3, 2, 4, 1} ; T= {}.

Principe :
Pour trier L, *retirer* l'élément *minimum* de L,
L = L - {MIN(L)};
et le placer *à la fin* de la liste triée T.
T= T ∪ {MIN(L)};
Recommencer l'opération jusqu'à ce que L soit vide.
La liste résultante T est triée...
Exemple
L= {-5, 3, 2, 4, 1}; T= {};
L= { 3, 2, 4, 1}; T= {-5};
L= {3, 2, 4 }; T= {-5, 1};
L= {3, 4 }; T= {-5, 1, 2};
L= {4}; T= {-5, 1, 2, 3};
L = {}; T= {-5, 3, 2, 4, 1};

Programme C :

```
void tri_selection(typelement a[], int N)
{
int i,j;
int mini; /* indice du minimum de la liste */
for (i=0; i<N-1; i++)
{
  mini=i;
  for (j=i+1; j<N; j++)
    if (a[j]<mini)
      mini=j;
  echange(&a[i],&a[mini]);
}
```

Complexité de l'algorithme.
La recherche du minimum d'une liste non triée de N éléments nécessite de parcourir toute la liste et N-1 comparaisons. La longueur de la liste diminue à chaque itération. Donc le nombre de comparaisons est donné par :
(N-1) + (N-2) + ... + 1, soit N (N-1) / 2 comparaisons.
L'algorithme naïf est en $O(N^2)$ dans tous les cas. S'il y a 1000 éléments, cela fait un million de comparaisons !

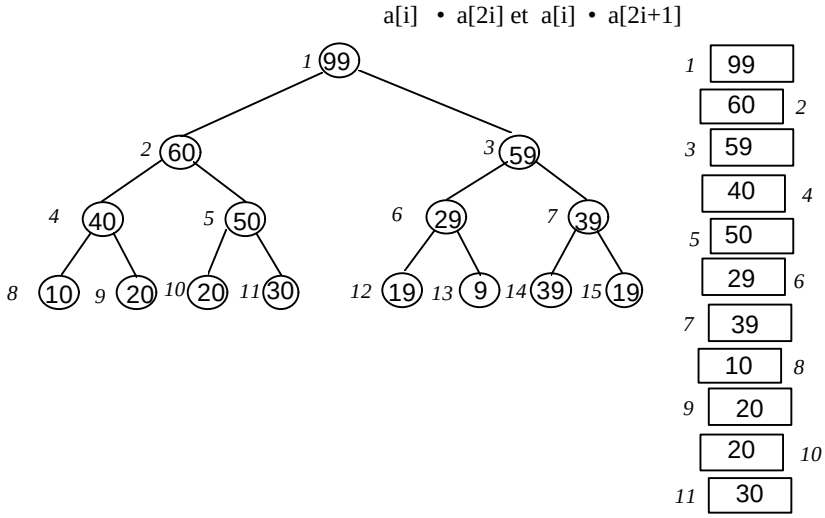
Cette complexité peut être améliorée si la structure de données utilisée permet une recherche accélérée de l'élément minimum. C'est le cas du tri maximier ou tri par tas.

Tri maximier ou tri par tas [*Heap sort*]
Le tri par tas est aussi un tri par sélection, sur place, dont l'efficacité est due à la structure de données astucieuse, un arbre maximier. Sa complexité est O(n log n).

Principe :
On opére en deux étapes :
1. On fabrique un tas [*heap*], c'est-à-dire un arbre binaire partiellement ordonné dit arbre "maximier" qui peut être représenté par un tableau tel que a[i] a[2i] et a[i] a[2i+1] pour 1 i N. Si on représente les éléments 2i et 2i+1 comme les fils du noeud i, alors *a* est un arbre binaire équilibré pour lequel la clé d'un noeud n'est jamais dépassée par les clés de ses descendants. Pour cela on tamise successivement a[N/2], ..., a[1].

2. Puis on extrait successivement le maximum de l'arbre, qui par construction se trouve en a[1] en l'échangeant avec a[M] pour M = N, N-1, ..., 1. Et comme cette opération a supprimé la propriété de tas, on la rétablit en tamisant le nouveau a[1].

Arbre maximier



Le tamisage

Le tamisage de a[k] consiste à échanger itérativement a[k] avec le maximum de a[2k] et de a[2k+1] jusqu'à ce que la condition de tas soit satisfaite.

La procédure tamiser(k, M) insére a[k] parmi a[k] ... a[M] :

Exemple :

	Tamisage			Heap sort								Liste triée	
1	24	24	**81**	*81*	***5***	**45**	*45*	**2**	*24*	***2***	**5**	*2*	2
2	5	**45**	45	45	45	**5**	5	5	5	5	**2**	*5*	5
3	81	81	**24**	24	24	24	24	**24**	*2*	***24***	24	24	24
4	2	2	2	2	2	2	*2*	45	45	45	45	45	45
5	45	**5**	5	*5*	***81***	81	81	81	81	81	81	81	81

Programme C :

```
void tamiser(int k; int M)
{
int j, v;
v = a[k];
while (k<= M/2)
{
   j = 2*k;
   if ((j<M) && (a[j+1] > a[j]))
      j++;
   if (v>=a[j])
      break;
   a[k] = a[j];
   k = j;
}
a[k] = v;
}

void heapsort(int N)
{
int k;
for (k= (N/2)-1; k>=0; k- -)
   tamiser(k, N);
for (k=N-1; k>0; k- -)
{
   echange(&a[0], &a[k]);
   tamiser(0,k);
}
}
```

Complexité :

O(n log n).

```
/* Tri du TAS ou tri maximier [Heap
sort] */
/* JF */
#include <stdio.h>
#include <stdlib.h>

#define MAXELEMENT 10

int T[MAXELEMENT + 1];
```

```
void echanger(int *a, int *b)
{
  int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}

void range_tas(int premier, int
dernier)
/* Suppose que T[premier], ...,
T[dernier] possedent la propriete des
maximiers, sauf peut-etre pour les
fils de T[premier] */
/* La procedure fait descendre
T[premier] jusqu'a restauration de
l'ordre partiel sur tout l'arbre */
{
  int r;
  r=premier;
  while (r <= dernier / 2)
  {
    if (dernier == 2*r)
    {
      if (T[r]>T[2*r])
        echanger(&T[r], &T[2*r]);
      r = dernier;
    }
    else if ((T[r]>T[2*r]) &&
(T[2*r]<=T[2*r+1]))
    {
      echanger(&T[r], &T[2*r]);
      r = 2*r;
    }
    else if ((T[r]>T[2*r+1]) &&
(T[2*r+1]<T[2*r]))
    {
      echanger(&T[r], &T[2*r+1]);
      r = 2*r+1;
    }
    else
      r = dernier;  /* sortie du
while */
  }
}


void TriTas(int n)
/* tri par tas de n elements ranges
dans le tableau T */
{
  int i;
  for (i=n /2; i>=1; i--)
    range_tas(i,n);
  for (i=n ; i>=2; i--)
  {
    echanger(&T[1], &T[i]);
    range_tas(1, i-1);
  }
}


/* ------------------------- MAIN ---
-------------- */
void main (void)
{
int i;
srand (1);
for (i=1; i<=MAXELEMENT; i++)
{
  T[i]=rand();
}

printf("\nLISTE A TRIER ------\n");
for (i = 1; i<=MAXELEMENT; i++)
  printf("%d ",T[i]);
printf("\n--------------------\n");

TriTas(MAXELEMENT);

printf("\nLISTE TRIEE  ------\n");
for (i = MAXELEMENT; i>=1; i--)
  printf("%d ",T[i]);
printf("\n--------------------\n");
}
```

Tri à bulle

Le tri à bulle (buble sort) consiste à comparer les éléments consécutifs et à les échanger si l'ordre recherché est violé (les bulles légéres remontent vers le haut).

Le tri peut se faire sur place, et la fin du tri survient quand plus aucun échange n'a eu lieu au cours d'une traversée de la liste.

Principe :

Faire venir en a[i] l'élément minimum de a[i], a[i+1], ..., a[N-1] par échange d'éléments adjacents.

Exemple :

0	**5**	2	2	2	2		**1**		**1**
1	**2**	**5**	1	1	1		**2**		**2**
2	1	**1**	**5**	3	3		3		**3**
3	3	3	**3**	**5**	4		4		**4**
4	4	4	4	**4**	5		5		**5**

Programme C :

```
for (i=0; i<N-1; i++)
    for(j=N-1; j>i; j--)
    {
        if (a[j]<a[j-1])
            echange(&a[j-1],&a[j]);
    }
```

Exercice
- Ecrire un algorithme de tri à bulle amélioré en détectant la fin du tri (aucun échange n'a eu lieu lors de la dernière passe) et en ne repassant pas sur les valeurs qui sont déjà ordonnées.
- Quelle est sa complexité dans le meilleur des cas, dans le pire des cas, et en moyenne ?

Tri par insertion

Principe : A la i-éme étape, insérer a[i] parmi a[1], a[2], ..., a[i-1] déjà triés.

Exemple :

Indice	*1*	*2*	*3*	*4*	
0	**5**	**2**	1	1	**1**
1	**2**	5	2	2	**2**
2	1	**1**	**5**	3	**3**
3	3	3	**3**	**5**	**4**
4	4	4	4	**4**	**5**

Programme C:

```
void tri-insertion(typelement a[], int N)
{
typelement v; /* une variable auxiliaire */
int i,j;
for (i=1; i<N; i++)
{
  v=a[i];
  j=i;
  while ((j>0) && (a[j-1]>v))
  {
    a[j]=a[j-1];
    j--;
  }
  t[j]=v;
}
}
```

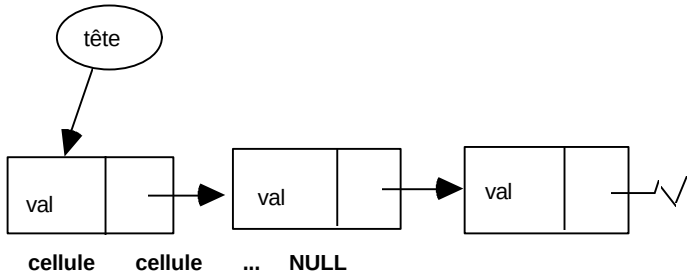
Complexité : $O(N^2)$ au pire des cas et en moyenne.

Tris sur les Listes

Une liste est une structure de données constituée de cellules chaînées les unes aux autres par pointeurs. Une cellule est un enregistrement :

```
typedef struct cellule {
  int val;
  struct cellule *suiv ;
  } *liste;
```

La tête de liste est désignée par un pointeur ***tête*** de type **liste** ; c'est une cellule qui est chaînée aux cellules suivantes par l'intermédiaire du pointeur ***suiv***.
La dernière cellule n'a pas de successeur. Elle pointe sur **NULL**.



On dira qu'une liste de valeurs $<n_1, n_2, ..., n_k>$ est triée si $n_1 <= n_2 <= ... <= n_k$.

Tri par insertion

- Ecrire une fonction **liste insere(liste l, int x)**

qui, si on l'appelle avec les paramètres ***x*** et ***l*** pointant sur la liste $<n_1, n_2, ..., n_k>$ **triée**, rend un pointeur sur une liste de la forme $<n_1, n_2, ...,n_i,$ ***x,*** $n_{i+1}, ..., n_k>$ elle aussi triée, et laissant intacte la liste ***l***. On écrira cette fonction de façon récursive.

Exercice 2 : Tri par insertion et modification

- Ecrire une fonction **void insere_modifie(liste l, int x)**

qui, si on l'appelle avec les paramètres ***x*** et ***l*** pointant sur la liste $<n_1, n_2, ..., n_k>$ **triée**, rend la liste ***l*** modifiée et triée de la forme $<n_1, n_2, ...,n_i$ ***,x,*** $n_{i+1}, ..., n_k>$. On écrira cette fonction de façon itérative. Justifier l'utilisation d'un pointeur sur ***l***..

Exercice 3 : Fonction de tri récursive

- Ecrire une fonction de tri : **liste tri_ins (liste l)**

qui fonctionne de la façon suivante : si son argument est vide, elle rend la liste vide. Sinon elle insére (au moyen de la fonction **insere**) la tête de son argument (premier entier dans la liste) dans la liste obtenue en triant (en utilisant **tri_ins**) la queue de son argument (suite de la liste passée en argument).

Exercice 4 : Fonction de tri itérative

Ecrire une fonction de tri :**liste tri_ins_iterative (liste l)**
de façon itérative qui utilise une liste auxiliaire, initialement vide, à laquelle elle ajoute successivement au moyen de la fonction **insere_modifie** tous les éléments de son argument, qu'elle parcourt jusqu'au bout. Puis elle retourne cette liste auxiliaire.

Exercice 5 : Complexité

- Que pensez-vous de ces deux fonctions de tri, du point de vue de la clarté des algorithmes (comment prouver leur correction) ?
- Du point de vue de la consommation d'espace mémoire ?
- Du point de vue du nombre de comparaisons ?

Shellsort

Le shell sort est un tri par insertion, mais au lieu de placer chaque élément à sa place dans la sous-liste triée de tous les éléments qui le précédent, on le place dans une sous-liste d'éléments qui le précédent distants d'un certain incrément ***incr***, que l'on diminue au cours de passages successifs sur la liste. Cela a pour effet de déplacer très rapidement les éléments très éloignés de leur position finale.

Au premier passage on crée des sous-listes d'éléments distants de *n/2* qu'on trie séparément ; au 2éme passage des sous-listes d'éléments distants de *n/4*, qu'on trie à leur tour séparément, etc. A chaque passage, l'incrément est divisé par deux et toutes les sous-listes triées par insertion séquentielle. Le tri s'arrête lorsque l'incrément est nul.

- Montrer que si deux éléments **t[***i* **]** et **t[***i*+*n*/2^k **]** ont été échangés au *kiéme* passage, alors ils restent triés par la suite.
- Programmer un shell-sort sur un tableau tiré aléatoirement de 1000 valeurs. Comparer avec le tri à bulle sur le même ensemble de données (nombre de comparaisons et d'échanges).

Tri par fusion

Ce tri procéde suivant le principe "diviser pour résoudre".

On doit disposer de deux fonctions récursives :

1) Découpage récursif

Une fonction récursive dont la déclaration est :

void decoupe(liste l, liste p, liste i)

telle qu'après l'appel sur des arguments ***l*** = <n_1, n_2, ..., n_k > , ***p***, ***i***, la variable ***p*** contienne la liste <n_2 ,n_4, ...> des éléments de rang pair de ***l***, et ***i*** contienne la liste <n_1, n_3, ...> des éléments de rang impair de ***l***..

2) Fusion

Une fonction dont la déclaration est :

liste fusion (liste n , liste m)

telle que le résultat de l'appel de fusion sur deux listes **triées** (hypothése essentielle) ***n*** = <n_1, n_2, ..., n_k > et ***m*** = <m_1, m_2, ..., m_p > soit une liste triée de longueur **k+p** contenant tous les éléments de ***n*** et de ***m*** (autrement dit une permutation triée de la liste <n_1, n_2, ..., n_k , m_1, m_2, ..., m_p >). Mais attention, on veut que le nombre de permutations réalisées pour effectuer cette opération soit borné par **k+p**.

Cette fonction pourra, au choix, être écrite de façon récursive ou itérative.

3) Tri par fusion

A l'aide des deux fonctions précédentes écrire la fonction récursive de tri par fusion :

liste tri_fusion (liste l)

dont le principe est le suivant : si l'argument est vide ou n'a qu'un seul élément, on retourne l'argument comme résultat. Sinon on découpe l'argument en deux sous-listes au moyen de **decoupe**, on rappelle **tri_fusion** sur chaque partie, et on obtient ainsi deux listes triées ***l*** et ***l'*** (expliquer pourquoi ?)

On applique la fonction **fusion** à ces deux listes et le résultat obtenu est, si **fusion** est correctement écrite, une liste triée.

Tri rapide [*quick-sort*]

Ce tri procéde aussi comme le précédent selon le principe "diviser pour résoudre". Il différe essentiellement par la phase "diviser";

1) Découpage ordonné
- Ecrire une fonction de découpage :
void decoupe_ordre (liste l, int x , liste i, liste s)
telle que, après appel sur les arguments ***l*** = <n_1, n_2, ..., n_k > , ***x*** , ***i***, ***s*** , la variable ***i*** contienne la liste des éléments qui sont inférieurs ou égaux à ***x,*** et la variable ***s*** contienne la liste des éléments de ***l*** qui sont strictement supérieurs à ***x.*** L'entier ***x*** est appelé *pivot* . Cette fonction opére donc une partition de ***l***.

2) Quick-sort.
- Ecrire une fonction récursive de tri :
liste tri_quick (liste l)
dont le principe est le suivant : si l'argument est vide, ou n'a qu'un seul élément, on rend l'argument comme résultat. Sinon, l'argument est de la forme <***n***, n_1, n_2, ..., n_k > avec ($k>=1$). On utilise la fonction **decoupe_ordre** pour partitionner la liste <n_1, n_2, ..., n_k > en utilisant ***n*** comme pivot. On obtient alors deux listes. Appelons les l_1 et l_2 . Tous les éléments de l_1 sont inférieurs ou égaux à *n*, et tous ceux de l_2 sont supérieurs à *n*.
On trie ces deux listes au moyen de **tri_quick** et on obtient deux listes triées l'_1 et l'_2 . On renvoie alors la concaténation de l'_1 et ***n*** .l'_2 qui est une liste triée.

Complexité

Les algorithmes de tri à bulle sont en O(n*n). L'algorithme quick-sort en O(n $\log_2$(n)).

Bibliographie

Informatique et calcul numérique

Breton Ph., Dufourd G, Heilman E., "Pour comprendre l'informatique",Hachette, 1992
Cohen J.H., Joutel F., Cordier Y., Jech B. "Turbo Pascal, initiation et applications scientifiques", Ellipses, 1989
Leygnac C., Thomas R. "Applications de l'informatique, études de thèmes en mathématiques, physique et chimie", Bréal, 1990
Piskounov N., “Calcul différentiel et intégral”, Editions Mir, 2 tomes.

Algorithmique

Aho A., Hopcroft J., Ulman J., “Structures de données et algorithmes” - InterEditions 1987
Beauquier D., Berstel J., Chretienne Ph., “Eléments d’algorithmique” - Masson 1992.
Carrez C. “Des structures aux bases de données” - Dunod 1990.
Crochemore M. “Méthodologie de la programmation et algorithmique” - UMLV 1990.
Perrin D. "Cours d'Informatique DEUG SSM Module M2" - UMLV 1997

Programmation

PATTIS R. E. “Karel the Robot, a gentle intrduction to the art of programming” - John Wilez & Sons, 1995.
Kerninghan B. , Ritchie D. "Le langage C" - Masson 1984
BORLAND, "Turbo C 2.0 "Manuel de l'utilisateur"
BORLAND, "Turbo C 2.0 "Manuel de référence"
Leblanc G., "Turbo C" - Eyrolles 1988
Charbonnel J., "Langage C - Les finesses d'un langage remarquable" - Armand Colin 1992

Table des matiéres