COURS
D'ALGORITHMIQUE II ET METHODES DE PROGRAMMATION

Par Ir. NSIAMUNU DAVID Jonathan

Licencié en sciences mathématiques et informatiques

UNIVERSITE DE KINSHASA

Les "astuces de programmation" sont une calamité.
Un bon programme dit ce qu'il fait et fait ce qu'il dit.

Ce syllabus s'adresse aux étudiants ayant peu ou prou des connaissances en informatique et il s'adresse tout particulièrement aux étudiants de premier cycle en informatique qui pourront appréhender les notions et les méthodes de programmation qu'ils aborderons dans leurs études ; ce cours constituent précisément une introduction à l'aspect scientifique de l'informatique ,environnements informatiques et à la science sous adjacente.

On étudiera ici principalement les ***algorithmes*** et ***structures des données***, i.e la structuration de l'information en machine, les plus communément employés en informatique, mais dont certains sont utiles dans d'autres domaines des sciences de l'ingénieur.

Le but de tel cours est triple :

- D'abord, faire comprendre que l'informatique ne se limite pas à la simple écriture de programmes, mais qu'elle nécessite une réflexion mathématique approfondie afin de déterminer le bon algorithme et la bonne structure de données permettant de résoudre le problème auquel on est confronté. Se faire l'idée sur la complexité (spatiale ou temporelle) d'un algorithme, il s'agit de l'espace mémoire et du temps nécessaire à son exécution.
- Ensuite, fournir à l'ingénieur une connaissance suffisante des principaux types d'algorithmes et de structures de données pour qu'il évite de réinventer la roue et puisse se référer aisément à ce qui existe déjà.
- Enfin, indiquer les limitations de l'usage de l'ordinateur : tous les problèmes ne sont solubles par l'ordinateur, ou bien à cause d'une complexité temporelle ou spatiale rédhibitoire des algorithmes qui leur correspondent, ou bien même parce qu'ils sont *indécidables*, c'est-à-dire réellement trop compliqués pour être résolus par un procédé informatique.

Nous avons choisi Java et C pour s'exercé à la programmation car ses sont des langages typé assez répandu qui permet de s'initier aux diverses constructions présentes dans la plupart des langages de programmation modernes.

A ces cours sont couples des séances de travaux dirigées et pratiques qui sont beaucoup plus qu'un complément au cours, puisque c'est en écrivant des programmes que l'on apprend l'informatique.

MODES D'EVALUATION

Le cours sera évalué de la manière suivante :

- Sous forme de travaux réalisés pendant l'année (Présence aux cours, aux T.P.)
- Sous forme d'exposés (Par groupe d'étudiants, les étudiants choisissent un thème, le développent et l'exposent avant les examens)
- Sous forme d'interrogation et d'examen oral et / ou écrit

CONTENU

- CHAPITRE I : STRUCTURE D'UN PROGRAMME INFORMATIQUE
- CHAPITRE II : LA RECURSIVITE
- CHAPITRE III : LES STRUCTURES DE DONNEES

CHAPITRE I : STRUCTURE D'UN PROGRAMME

INTRODUCTION

L'algorithmique est un ensemble des méthodes permettant de définir et/ou d'étudier des algorithmes et Algorithme est une séquence finie d'actions permettant de résoudre un problème donné.

Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation; mais lorsqu'on programme dans un langage (en C, en Visual Basic, etc.) on doit en plus se colleter les problèmes de syntaxe, ou de types d'instructions, propres à ce langage. Apprendre l'algorithmique de manière séparée, c'est donc sérier les difficultés pour mieux les vaincre.

La maîtrise de l'algorithmique requiert deux qualités, très complémentaires d'ailleurs :

- il faut avoir une certaine intuition, car aucune recette ne permet de savoir a priori quelles instructions permettront d'obtenir le résultat voulu. C'est là, si l'on y tient, qu'intervient la forme « d'intelligence » requise pour l'algorithmique. Alors, c'est certain, il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, et j'insiste sur ce point, les réflexes, cela s'acquiert. Et ce qu'on appelle l'intuition n'est finalement que de l'expérience tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».

- il faut être méthodique et rigoureux. En effet, chaque fois qu'on écrit une série d'instructions qu'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter, armé d'un papier et d'un crayon, afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas la moindre once d'intelligence. Mais elle reste néanmoins indispensable, si l'on ne veut pas écrire à l'aveuglette.

Le but de l'algorithmique

Le but de ce cours est donc de vous apprendre à créer des algorithmes, c'est-à-dire à décomposer des calculs compliqués en successions d'étapes simples mais indépendamment d'un langage de programmation.

Chaque fois qu'il nous arrivera de résoudre un problème, nous présenterons non pas la solution du problème mais une solution au problème. Le problème du choix du meilleur algorithme est traité par l'algorithme qui prend en compte les critères tels que le temps d'exécution par le processeur et l'utilisation de la mémoire (complexité d'un algorithme).

Les langages de programmation permettent de créer des programmes, des applications mobiles, des sites Internet, des systèmes d'exploitation, etc. Certains langages de programmation sont extrêmement populaires. En mai 2012, les 10 langages de programmation les plus populaires étaient le C, le Java, le C++, l 'Objective-C, le C#, le PHP, le Basic, le Python, le Perl et le JavaScript.

La programmation est une démarche à deux étapes essentielles à savoir :

- La résolution du problème qui commence par la mise sur pied d'un algorithme, suite d'instructions décrivant les étapes du processus de traitement indépendamment d'un quelconque langage de programmation ;

- La codification de l'algorithme, dans un langage de programmation donné, qui produit un programme.

Nous ferons donc dans ce cours abstraction du langage utilisé. Nous nous intéresserons uniquement à la façon de combiner des instructions pour former des programmes, indépendamment des langages de programmation.

Retenons donc que sous le vocable programmation se cachent deux activités essentielles : l'analyse et la codification. La première est de loin la plus difficile et la plus importante.

I.1 squelette d'un programme informatique

Un programme informatique se présente généralement de la manière suivante :

{En-tête du programme}
Déclaration des variables
<Suite d'instructions (corps du programme)>
Fin

- L'en-tête du programme est constitué du mot clé PROGRAM suivi du nom donné au programme, pour certains langages de programmation, cet en-tête facultatif.
- La zone de déclaration des variables : les variables sont déclarées comme expliqué ci-dessus. Quand il s'agit des variables de type composé, la déclaration de type précédera celle des variables.
- Le corps du programme constitue d'une suite d'instructions (lignes du programme séparées, généralement par un point-virgule (;) quand le programme comporte un (des) modules(s) (procédure, fonction), celui-ci (ceux-ci) se place(nt) tout juste après la zone de déclaration des variables.
- La fin du programme généralement par le mot-clé Fin suivi d'un point(.)

NB : Un programme manipule des données caractérisées par un nom et un type. Ces données sont stockées en mémoire. Au moment de la traduction du programme, le compilateur affecte à chaque donnée un emplacement en mémoire caractérisé par une adresse et une taille.

I.2 Notion de type de données

Puisqu'une mémoire d'ordinateur ne peut contenir que des données binaires, il faut maintenant décider comment représenter des informations en mémoire. On s'appuie pour cela sur la notion de type de données, qui définit comment une certaine catégorie d'information est représentée en mémoire, comment on y accède et comment on la modifie.

Type de données: Un type en programmation précise l'ensemble des valeurs que peut prendre une variable; les opérations que l'on peut effectuer sur une variable dépendent de son type.

Comme nous l'avons vu précédemment, un identificateur est un nom permettant de désigner facilement un concept dans un programme (et ici, ce concept est un type). Par exemple, le langage C contient un type appelé int et utilisé pour représenter des nombres entiers.

Les règles de représentation décrivent la forme que l'information doit prendre en mémoire. Cette forme est appelée un code. La règle permettant de transformer l'information en un code est appelée le codage.

Code: représentation d'une information en mémoire, conformément aux règles définies pour un type donné.

On distingue deux sortes de types : les types simples et les types complexes.

I.2.1 les types élémentaires ou simples

Type de données décrivant une information atomique.

a) Les types numériques

Le type numérique est formé de l'ensemble des nombres que le processeur peut manipuler. On distingue :

- Le type entier(integer) : associé aux nombres sans partie décimale ;
- Le type réel (real) : associé aux nombres ayant une partie décimale ou partie fractionnaire.

b) Le type caractère(character) : le type est formé de l'ensemble des caractères que le processeur reconnaît. La valeur d'une variable de type caractère sera mise entre cote.

c) Le type chaîne de caractère (chain of character) : une chaîne de caractère est formée par une juxtaposition des caractères.

d) Le type logique ou booléen : une variable du type logique ne peut que prendre l'une des deux valeurs vraie ou fausse que l'on représente parfois respectivement par 0 et 1.

I.2.2 Types composés

Un type composé résulte de l'association de plusieurs types élémentaires.

Ex : les informations concernant un étudiant peuvent être rangées dans une variable composées ***étudiant*** constituée des variables élémentaires *nom* (de type chaîne), *âge* (de type numérique (entier)) et *sexe* (de type caractère).

Nom,*âge* et *sexe* sont les subdivisions de la variable composée ***étudiant*** qui seront respectivement désignées par étudiant.nom,étudiant.âge et étudiant.sexe, une variable composée dont les différentes subdivisions ne sont pas de même type est appelée un *enregistrement* (record) par opposition à un *tableau* (array) dont toutes les subdivisions sont de même type.

Selon la richesse d'un langage, on peut construire de nombreux types spéciaux tels que le type enregistrement, le type d'ensemble, le type énuméré, le type intervalle…, dans ce cas, la définition de ces types précède la déclaration des variables.

Exemple :

```
Type student=record
        Nom : chaîne
       Age : entier
Var x,y : student
I :entier
Début
{Suite d'instructions}
 Fin
```

1.3 Les variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou que sais-je encore. Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs.

Ces données peuvent être de plusieurs types (on en reparlera) : elles peuvent être des nombres, du texte, etc.
Toujours est-il que dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une variable.

Pour employer une image, *une variable* est une boîte, que le programme (l'ordinateur) va repérer par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.

1.4 Valeur d'une variable :

Contenu de la variable, i.e. Séquence des bits présents dans le morceau de mémoire associé à la variable, interprétée en utilisant le type de la variable.

1.4.1 Déclaration des variables

Pour créer une variable, il est nécessaire de la déclarer :

La première chose à faire avant de pouvoir utiliser une variable est de créer la boîte et de lui coller une étiquette. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle *la déclaration des variables*.

Déclaration d'une variable: action de définir une variable en précisant son identificateur et son type.
Adresse d'une variable : adresse du premier octet du morceau de mémoire alloué à la variable.
Allocation : action de réserver un morceau de mémoire pour une utilisation particulière.
Désallocation : opération inverse de l'allocation, consistant à libérer une partie de la mémoire qui était jusque-là réservée à une utilisation spécifique.

Le nom de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages. Toutefois, une règle absolue est qu'un nom de variable peut comporter des lettres et des chiffres, mais qu'il exclut la plupart des signes de ponctuation, en particulier les espaces. Un nom de variable correct commence également impérativement par une lettre. Quant au nombre maximal de signes pour un nom de variable, il dépend du langage utilisé.

Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; encore doit-on préciser ce que l'on voudra mettre dedans, car de cela dépendent *la taille* de la boîte (de l'emplacement mémoire) et *le type* de codage utilisé.

1.4.2 Types des variables

1.4.2.1 Types numériques

Tous les langages, quels qu'ils soient offrent un « bouquet » de types numériques, dont le détail est susceptible de varier légèrement d'un langage à l'autre. Grosso modo, on retrouve cependant les types suivants :

Type Numérique	Plage
Byte (octet)	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3,40x10^{38}$ à $-1,40x10^{45}$ pour les valeurs négatives $1,40x10^{-45}$ à $3,40x10^{38}$ pour les valeurs positives
Réel double	$1,79x10^{308}$ à $-4,94x10^{-324}$ pour les valeurs négatives $4,94x10^{-324}$ à $1,79x10^{308}$ pour les valeurs positives

Autres types numériques : Certains langages autorisent d'autres types numériques, notamment : le type *monétaire* (avec strictement deux chiffres après la virgule) le type *date* (jour/mois/année).

1.4.2.2 Type alphanumérique

On dispose donc également du type *alphanumérique* (également appelé type *caractère*, type chaîne ou en anglais, le type *string*, Dans une variable de ce type, on stocke des caractères, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou même de chiffres. Le nombre maximal de caractères pouvant être stockés dans une seule variable string dépend du langage utilisé.

1.4.2.3 Type booléen

Le dernier type de variables est le type *booléen* : on y stocke uniquement les valeurs logiques VRAI et FAUX.

1.5 L'action Lire

Cette action permet d'effectuer à une variable de l'environnement une valeur frappée au clavier.
Exemple : Ecrire un algorithme permettant de lire une valeur entière saisi au clavier.

```
Début
Var a : entier
Lire a
Fin
```

1.6 L'action Afficher

L'action afficher permet de faire apparaître une information à l'écran, produit comme effet l'affichage à l'écran de la valeur de l'objet.

Exemple : Ecrire un algorithme permettant d'afficher une valeur entière saisi au clavier.

```
Début
Var a : entier
Lire a
Afficher (« la valeur est », a)
Fin
```

1.7 L'action affectée

L'affectation permet de ranger une valeur dans un objet. Contrairement à l'action lire qui permet d'attribuer une valeur à un objet de manière externe c'est-à-dire tapant la valeur au clavier. L'affectation permet de le faire de manière interne.

Le format de l'action Affecter est : v x◄——

Exemple : Ecrire un algorithme permettant d'affecter une valeur entière.

```
Début
Var a : entier
a ◄——3
Afficher (« la valeur est », a)
Fin
```

1.7 Les opérateurs

Une expression est un ensemble de valeurs, reliées par des opérateurs, et équivalent à une seule valeur et *un opérateur* est un signe qui relie deux valeurs, pour produire un résultat. Parmi les opérateurs nous pouvons citez : les Opérateurs numériques, l'opérateur alphanumérique, les opérateurs logiques (ou booléens).

Nature	Variables utilisées	Notation	Signification
Opérateurs arithmétiques	Entier Réel	+	Addition
		-	Soustraction
		*	Multiplication
		/	Division (réelle)
		DIV	Division entière
		MOD	Reste de la division entière
Opérateurs logiques	Booléen Entier	et	Fonction ET
		ou	Fonction OU
		ouex	Fonction OU EXCLUSIF
		non	Fonction NON
Opérateur de concaténation	Chaîne de caractères	+	Concaténation
Opérateurs de comparaison	Booléen Entier Réel Caractère Chaîne de caractères	=	Egal
		≠	Différent
		<	Inférieur
		>	Supérieur
		≤	Inférieur ou égal
		≥	Supérieur ou égal

N div p donne la partie entière du quotient de la division entière de n par p, tandis que n mod p donne le reste de la division entière de n par p.

1.8 Ordres de priorité

Il existe trois ordres de priorité pour les opérations arithmétiques. La plus grande priorité revient l'exponentiation suivi de la multiplication et la division, et la plus faible priorité à l'addition et la soustraction. Mais lorsque deux opérations ont la même priorité, le processeur commencera par celle de gauche. Les opérations entre parenthèses sont prioritaires sur les autres et c'est le couple de parenthèses le plus interne qui est traité le premier.

A titre d'exemple, notons l'ordre d'exécution de chacune des opérations de l'expression suivante : a/b +(c*f) *(d**2), on peut ainsi construire une représentation (arbre) dont les feuilles (extrémités) représentent les opérandes et les nœuds internes les opérateurs.

1.9 Concaténation

La concaténation notée // est une opération qui fusionne deux chaînes de caractères.
Exemple : « mal » // « heureux » produit la chaîne « malheureux ».

1.10 Commentaires

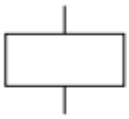
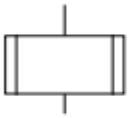
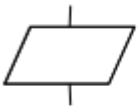
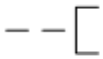
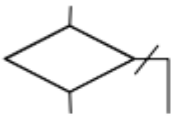
Les commentaires sont les lignes que l'on peut ajouter à un algorithme ou un programme pour le rendre plus compréhensible. Les commentaires n'affectent pas le programme et leur utilisation est purement optionnelle. Le traducteur ne s'occupe pas des commentaires d'autant plus qu'ils se sont pas des instructions exécutables. Nous conviendrons de mettre chaque ligne de commentaire entre *…* ou « … ».

1.11 Structure de contrôle et ordinogrammes

a) Construction des ordinogrammes

Au fur et à mesure qu'un algorithme devient complexe, il devient de plus en plus utile de représenter graphiquement ses étapes ainsi que le passage du contrôle entre ces étapes, tel est le but de l'*ordinogramme* ou *organigramme*. Dans un ordinogramme, chaque instruction est représentée dans une *boite* dont la forme est déterminée par le type d'opération. Le passage du contrôle entre ces boites sera représenté par des lignes *fléchées*. Si l'ordinogramme continue sur une autre page ou s'il est difficile de connecter deux boites, on utilisera un petit *cercle* avec une étiquette pour représenter cette connexion. D'où un *ordinogramme* est une représentation graphique d'un algorithme visant à mettre en valeur sa structure.

Quelques symboles utilisés pour construire un ordinogramme :

SYMBOLE	DESIGNATION	SYMBOLE	DESIGNATION
Symboles de traitement		Symboles auxiliaires	
	**Symbole général** Opération ou groupe d'opérations sur des données, instructions, pour laquelle il n'existe aucun symbole normalisé.		**Renvoi** Symbole utilisé deux fois pour assurer la continuité lorsqu'une partie de ligne de liaison n'est pas représentée.
	**Sous-programme** Portion de programme considérée comme une simple opération.		**Début, fin , interruption** Début, fin ou interruption d'un algorigramme.
	**Entrée-Sortie** Mise à disposition d'une information à traiter ou enregistrement d'une information traitée.		**Commentaire** Symbole utilisé pour donner des indications sur les opérations effectuées.
Symbole de test		Les différents symboles sont reliés entre eux par des lignes de liaisons.	
	**Branchement** Exploitation de conditions variables impliquant un choix parmi plusieurs.		
Sens conventionnel des liaisons			
Le sens général des lignes de liaison doit être : • De haut en bas • De gauche à droite Lorsque le sens général ne peut pas être respecté, des pointes de flèche à cheval sur la ligne indiquent le sens utilisé.			

b) **Structure de contrôle** :

Par structure de contrôle nous entendrons l'organisation des opérations dans un algorithme. La structure de contrôle détermine l'ordre dans lequel les opérations doivent être effectuées. Nous pouvons parmi les structures de contrôle la séquence, les schémas conditionnels, les itérations et les sous programmes.

.1. La séquence

La séquence est une suite d'actions dont l'exécution commence par la première et pas à pas, passe à toutes les instructions jusqu'à la dernière. Le contrôle passe d'une instruction à l'autre selon l'ordre dans lequel elles apparaissent dans l'algorithme. Toutefois, il arrive que la séquence soit interrompue pour une raison ou une autre pour poursuivre l'exécution à un autre endroit spécifique de l'algorithme,

on parle dans ce cas de *branchement* ou *transfert*. *Un branchement* est un transfert du contrôle de l'algorithme vers un endroit spécifique. Il existe deux types de branchements :

Branchement conditionnel

Le contrôle est transféré à l'une des lignes de l'algorithme moyennant la vérification d'une condition.

Condition : expression dont la valeur est interprétée de façon booléenne : la condition peut être soit vraie, soit fausse.

Exemple considérons l'algorithme suivant :

Si condition Alors
Instructions 1
Sinon
Instructions 2
Finsi

Avec le test simple, on associe une seule condition à un seul bloc de code. Si la condition est vraie, ce bloc est exécuté. Sinon, il n'est pas exécuté.

Branchement inconditionnel

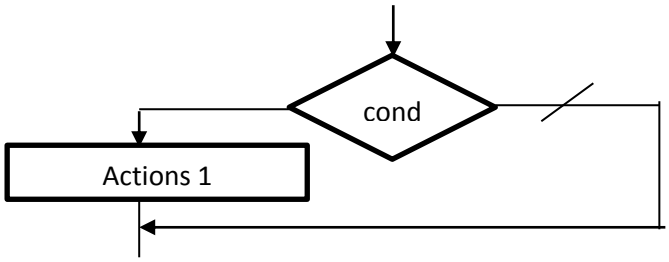
Le transfert est transféré inconditionnellement à une ligne spécifique de l'algorithme ou L'instruction *aller* renvoie l'exécution du programme vers un point spécifique repéré par une étiquette *n*.sa forme générale est *aller* à *n*, où n est une étiquette pour transférer le contrôle à la partie de l'algorithme qui commence par l'instruction étiquetée n.

Exemple : supposons que l'on veuille écrire un algorithme qui permet de multiplier un nombre x donné par 9 à l'aide d'un processeur n'ayant que l'addition comme action élémentaire.il s'agira en fait de trouver
P=a+a+a+a+a+a+a+a+a

Début
I=1
P=0
Lire x
4 p = p+x
I = i+1
Si i ≤ 9 alors aller à 4
Finsi
Afficher « le produit est : », p
Fin
Aller à 4 est un branchement conditionnel car le transfert à l'instruction 4 n'est réalisé que lorsque i ≤ 9. Par contre, le branchement serait inconditionnel.

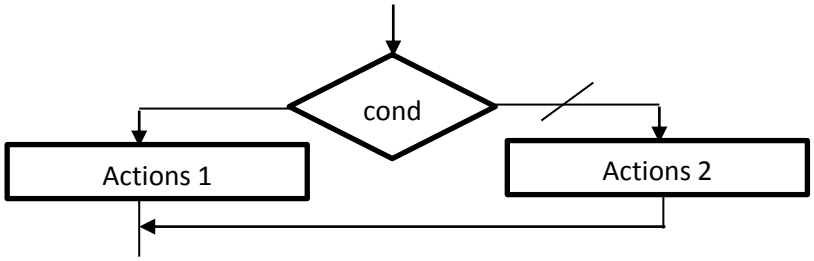
2 Les schémas conditionnels

A la différence du branchement inconditionnel, ici une condition est préalablement testée, et c'est le résultat du test qui détermine vers où transférer le contrôle.



Si condition *alors*
Action 1
Finsi
($1^{ère}$ $forme$)

Si condition *alors*
Action 1
Sinon
Action 2
Finsi
($2^{ère}$ $forme$)



Si la condition est vérifiée, l'action 1 est effectuée dans les deux cas. Si elle est fausse, dans le second cas on effectue l'action2. Le délimiteur *finsi* indique la fin de la séquence d'actions relatives à la partie *alors* pour la $1^{ère}$ $forme$. Pour la $2^{ère}$ $forme$, cette fin est délimitée par le mot *sinon*.

Exemple : Ecrire un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif ou positif (on laisse de côté le cas où le produit est nul). Attention toutefois : on ne doit pas calculer le produit des deux nombres.

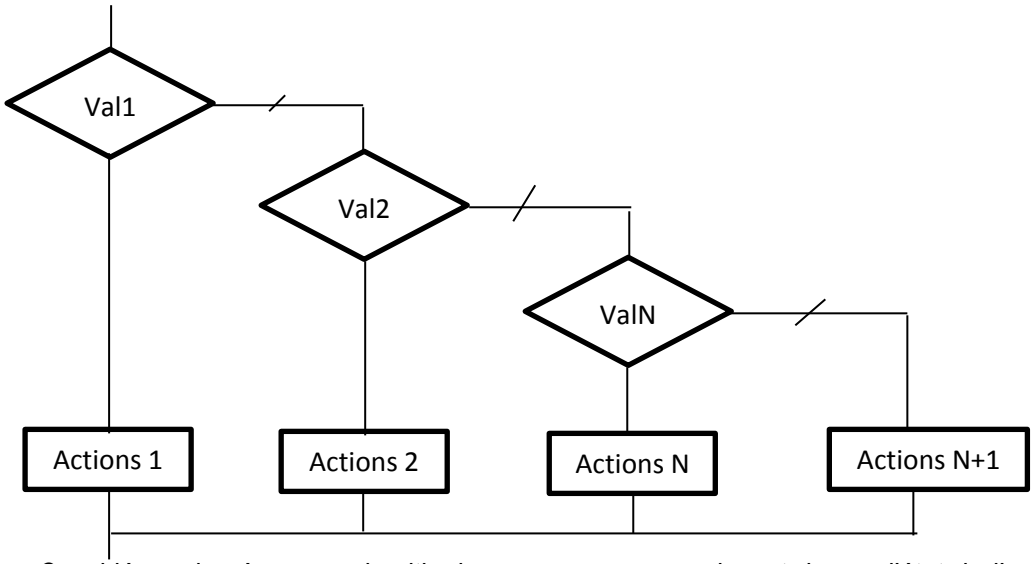
Corrigé :

Variables x, y en Entier
Début
Ecrire" Entrez deux nombres : "
Lire x, y
Si (x > 0 ET y > 0) OU (x < 0 ET y < 0) Alors
Ecrire "Leur produit est positif"
Sinon
Ecrire "Leur produit est négatif"
Finsi
Fin

Enoncé conditionnel généralisé

L'emboitement de plusieurs schémas conditionnels entraîne souvent l'écriture de séquences algorithmiques lourdes. Cette structure de choix multiple, une donnée est comparée successivement à des valeurs constantes.

Exemple : donnée = valx



Exemple : Considérons la séquence algorithmique, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).

```
Variable Temp en Entier
Début
Ecrire "Entrez la température de l'eau :"
Lire Temp
Si Temp =< 0 Alors
Ecrire "C'est de la glace"
Sinon
Si Temp < 100 Alors
Ecrire "C'est du liquide"
Sinon
Ecrire "C'est de la vapeur"
Finsi
Finsi
Fin
```

NB : Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes, Cette structure « *si* » successifs ou imbriqués avec trop grande imbrication est moins claire, c'est ainsi qu'au-delà de 3 possibilités (plusieurs alternatives), on recourt à la structure *cas… fincas*, qui permet de faire la sélection des instructions à exécuter selon le cas correspondant dans la liste.

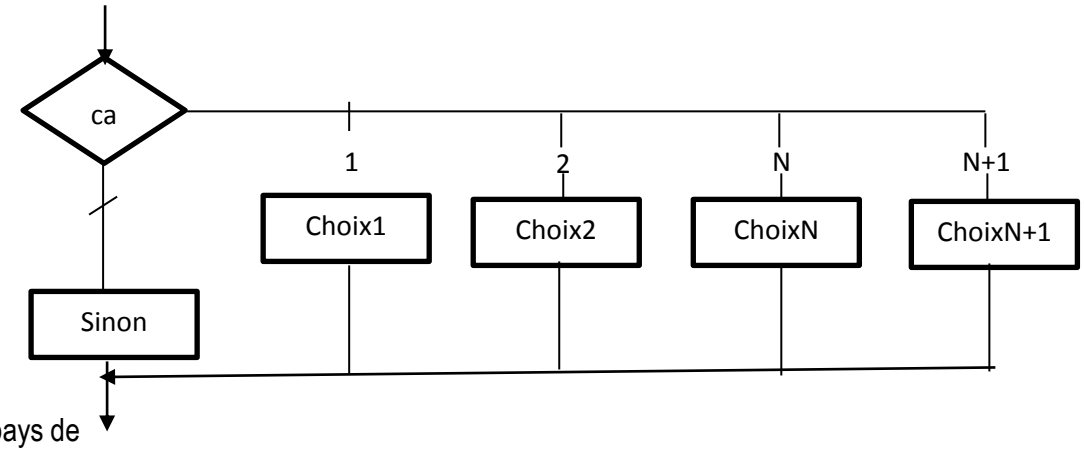
```
Suivant < variable > faire
    Cas < valeur1> < instructions1>
    Cas < valeur2><instructions2>
    …
   Cas< valeur n><instructions n>
   Autres cas<instructions>
FinSelon
```

Selon la valeur que prend la variable <variable>, le bloc d'instructions à exécuter est sélectionné. Par exemple, si la valeur de <variable> est <valeur 1>, alors le bloc <instructions 1> est exécuté. Le bloc <autres cas> est exécuté si la valeur de <variable> ne correspond à aucune des valeurs énumérées.

Considérons la séquence algorithmique suivante qui donne les capitales des pays de l'Afrique centrale.

```
Si pays= « RDC »
Alors capitale= « Kinshasa »
        Sinon si pays= « Congo »
                Alors capitale= « Brazzaville »
        Sinon si pays= « Gabon »
                Alors capitale= « Libreville »
        Sinon si pays= « RCA »
                Alors capitale= « Bangui »
         Sinon afficher « erreur »
      Finsi
Finsi
       Finsi
Finsi
```

Comme il Ya des situations mutuellement exclusives des « *si* », on utilisera le schéma généralisé *cas…fincas*.




```
Cas pays de
RDC : capitale = « kinshasa »
Congo : capitale = « brazzaville »
Gabon : capitale = « libreville »
RCA : capitale = « bangui »
Cameroun : capitale = « yaoundé »
Guinée Equatoriale : capitale = « malabo »
Sinon afficher « erreur »
 Fincas
```

NB : cette structure peut être utilisée dans le cas de l'intervalle
Par exemple : cas de val1 à val2 faire.

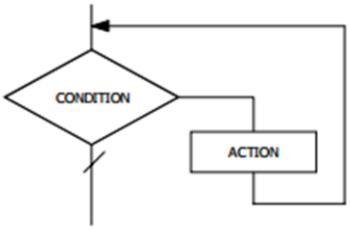
3 Les Bouclés

Une boucle permet d'exécuter plusieurs fois de suite une même séquence d'instructions. Cet ensemble d'instructions s'appelle le corps de la boucle. Chaque exécution du corps d'une boucle s'appelle une itération, ou encore un passage dans la boucle. Il existe trois types de boucle :

- Tant que
- Répéter ... jusqu'à
- Pour

➢ Le schéma Tant que

La syntaxe d'une boucle *Tant que* est la suivante.



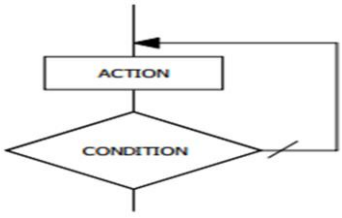
Tant que < condition >
< Instructions >
Fin tant que

La condition est évaluée *Avant* chaque passage dans la boucle, à chaque fois qu'elle est vérifiée, on exécute les instructions de la boucle. Une fois la condition n'est plus vérifiée, l'exécution se poursuit après la *fin tant que*. Affichons par exemple tous les nombres de 1 à 5 dans l'ordre croissant :

```
Algorithme : 1 à 5 Tant que
Var  i : numérique
Début
i←1
  Tant que i≤ 5
     Afficher i
      i ←i+1
  fin tant que
fin
```

Cet algorithme **initialise** i à 1 et tant que la valeur de i n'excède pas 5, cette valeur est affichée puis incrémentée. Les instructions se trouvant dans le corps de la boucle sont donc exécutées 5 fois de suite. La variable i s'appelle un **compteur**, on gère la boucle par incrémentations successives de i et on sort de la boucle une fois que i a atteint une certaine valeur. **L'initialisation du compteur est très importante !** Si vous n'initialisez pas i explicitement, alors cette variable contiendra n'importe quelle valeur et votre algorithme ne se comportera pas du tout comme prévu.

➢ Le schéma Répéter… jusqu'à ce que



Répéter
< instructions >
Jusqu'à ce que < condition >

Le fonctionnement est analogue à celui de la boucle tant que à quelques détails près :

- La condition est évaluée après chaque passage dans la boucle.
- On exécute le corps de la boucle jusqu'à ce que la condition soit vérifiée, donc tant que la condition est fausse.

Une boucle *Répéter ... jusqu'à ce que* est donc exécutée donc **au moins une fois**. Reprenons l'exemple précédent avec une boucle *Répéter ... jusqu'à* :

```
Algorithme : 1 à 5
Var i : numérique
Début
i←1
  répéter
      Afficher i
       i ←i+1
  jusqu'à ce que i > 5
fin
```

De la même façon que pour la boucle Tant que, le compteur est initialisé avant le premier passage dans la boucle. Par contre, la condition de sortie de la boucle n'est pas la même, on ne sort de la boucle qu'une fois que la valeur 5 a été affichée. Or, i est incrémenté après l'affichage, par conséquent i aura la valeur 6 à la n de l'itération pendant laquelle la valeur 5 aura été affichée. C'est pour cela qu'on ne sort de la boucle qu'une fois que i a dépassé strictement la valeur 5. Un des usages les plus courants de la boucle *Répéter ... jusqu'à* ce que est le contrôle de saisie :

- **Le schéma Pour**

Pour < variable> allant de < première valeur> à < dernière valeur> [par pas de <pas>]
< Instructions >
Fin pour

La boucle Pour fait varier la valeur du compteur <variable> entre <première valeur et <dernière valeur>. Le <pas> est optionnel et permet de préciser la variation du compteur entre chaque itération, le pas par défaut est 1 et correspond donc à une incrémentation. Toute boucle *pour* peut-être réécrite avec une boucle *tant que*.

La boucle pour initialise le compteur <variable> à la <première valeur>, et tant que la dernière valeur n'a pas été atteinte, les <instructions> sont exécutées et le compteur incrémenté de <pas> si le pas est positif, de -<pas> si le pas est négatif.

```
Algorithme : 1 à 5 Pour
Var : i numérique
Début
Pour i allant de 1 à 5
  Afficher i
Fin pour
fin
```

Note : bien que l'on utilise une boucle *pour* quand on sait en rentrant dans la boucle combien d'itérations devront être faites. Par exemple, n'utilisez pas une boucle *pour* pour contrôler une saisie !

1.12 Fonctions de la bibliothèque

On appelle fonctions de la bibliothèque d'un langage L des fonctions incorporées au compilateur de ce langage du fait de leur utilisation fréquente.
C'est le cas des fonctions mod, sqrt, abs, int, cos, sin, pow, etc. dans la plupart des langages de programmation.

1.13 Les fonctions de texte

Une catégorie privilégiée de fonctions est celle qui nous permet de manipuler des chaînes de caractères. Nous avons déjà vu qu'on pouvait facilement « coller » deux chaînes l'une à l'autre avec l'opérateur de concaténation +. Mais ce que nous ne pouvions pas faire, et qui va être maintenant possible, c'est pratiquer des extractions de chaînes (moins douloureuses, il faut le noter, que les extractions dentaires).

Tous les langages, je dis bien tous, proposent peu ou prou les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

- Length (chaîne) : renvoie le nombre de caractères d'une chaîne
- Left(chaîne,n): renvoie les n caractères les plus à gauche dans chaîne.
- Right(chaîne,n): renvoie les n caractères les plus à droite dans chaîne
- Trouve (chaîne1, chaîne2) : renvoie un nombre correspondant à la position de chaîne2 dans chaîne1. Si chaîne2 n'est pas comprise dans chaîne1, la fonction renvoie zéro.
- Split : renvoie un tableau de chaînes qui contient les sous chaînes limitées par les caractères spécifiés.
- Contains : renvoie une valeur qui indique si objet spécifié apparaît dans une chaîne.
- Substring : récupère une sous chaîne, la sous chaîne démarre à une position de caractère spécifié.
- Mid : récupère une sous chaîne dans une chaine.
- Aléatoire : renvoie une valeur aléatoirement dans une intervalle.
- Ect…

Exercices

Objectifs : approfondir les notions vues (boucles, conditions, entrées / sorties, …)

1. Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de voyelles contenues dans cette phrase.

2. Écrire un algorithme qui calcule les racines réelles (si elles existent) d'un polynôme du second degré décrit par 3 coefficients réels a, b et c. Les solutions seront affichées à l'écran.

3. Un magasin de reprographie facture 100 Fc les dix premières photocopies, 90Fc les vingt suivantes et 80Fc au-delà. Ecrivez un algorithme qui demande à l'utilisateur le nombre de photocopies effectuées et qui affiche la facture correspondante.

4. Ecrivez un algorithme permettant d'afficher la factorielle d'un nombre entier.

5. Ecrivez un algorithme permettant d'apprécier l'indice de pignet (IP) d'une personne, à partir des éléments ci-après :(T) : la taille en cm, (P) : le poids en Kgs et (PT) : le périmètre thoracique en cm et ai constante ce qui suit : IP=T- (P+PT) et l'algorithme interprète les résultats suivants :
   - Apte à tous travaux si IP se situe entre 0 à 15
   - Apte aux travaux moyens si IP se situe entre 15 à 25
   - Apte aux travaux légers si IP se situe entre 25 à 35
   - Inapte de 35 et plus.

6. Ecrivez un algorithme qui demande une phrase à l'utilisateur et qui affiche à l'écran le nombre de mots de cette phrase. On suppose que les mots ne sont séparés que par des espaces.

7. Ecrivez un algorithme qui demande une phrase à l'utilisateur. Celui-ci entrera ensuite le rang d'un caractère à supprimer, et la nouvelle phrase doit être affichée (on doit réellement supprimer le caractère dans la variable qui stocke la phrase, et pas uniquement à l'écran).

8. Un texte peut cacher, dans l'ordre, les lettres d'un mot. Par exemple : « Marie est revenue ici » contient le mot « merci ». Ecrire un algorithme qui détermine si un texte proposé contient ou non un mot donné.

9. Ecrivez un algorithme qui demande une phrase T à l'utilisateur. Celui-ci entrera ensuite le segment de caractère à supprimer qui commence en position K et dont la longueur est L dans un texte T, et la nouvelle phrase doit être affichée.

10. Ecrire un algorithme qui permette de dire si un mot donnée est un polindrome, c'est-à-dire identique à son miroir.

    Exemple : esse,maxam,kayak,…

7) Faites un ordinogramme relatif à un système d'alarme qui s'annonce : si quelqu'un franchit la porte ou une fenêtre de la maison, et si l'alarme est active à ce moment-là, l'alarme sonore se déclenche ; l'alarme s'arrête lorsque l'on désactive le système d'alarme.

8) Que fait l'algorithme ci-après et designer son organigramme:

```
Var tst : entier =0
Début
Var  mp : chaines
Faire
Ecrire(" Entrez le code   ")
Lire mp
Si tst=3 alors
Quitter
Finsi
Si(non(mp= "david") alors
Ecrire(" incorrecte  ",(3-tst) & "essai")
Finsi
Tst+=1
Jusqu'à ce que (non(mp="david"))
Fin
```

9) Écrire un algorithme permettant de trouver une valeur choisie aléatoirement par le programme. Le joueur disposera au maximum de 3 tentatives pour trouver cette valeur et le programme lui indiquera à chaque essai si sa valeur est trop grande ou trop petite.

10) Écrire un algorithme permettant de savoir si une année saisie par l'utilisateur est bissextile ou non. Rappel : une année est bissextile si elle est divisible par 4 mais non divisible par 100.

11) Écrire un algorithme permettant de lire 10 nombres au clavier et d'afficher le carré des nombres pairs uniquement. Attention, on ne mémorisera pas les 10 valeurs saisies.

12) Écrire un algorithme permettant de calculer la somme des n premiers nombres impairs. Quel lien pouvez-vous établir entre la valeur obtenue et n ?

13) Écrire un algorithme permettant de calculer la somme des n premières puissances de 2.
Exemple : valeur saisie 5 résultat 63 (= 1+2+4+8+16+32)

14) Que fait cet algorithme

```
Début
Var n : entier
Afficher ("introduire la valeur entière   ")
Lire n
Faire
Si n % 2=0 alors
n/=2
Sinon
n=n*3+1
finsi
afficher ("" , n)
```

Jusqu'à tant que (n>1)
Fin

15) Quelles sont les résultats successives prises pour chaque variable pendant l'exécution de l'algorithme ci-après, lorsque la valeur lue sont : 4, 8 ; 30 ,14 ,17 et 27 ?

Début

Var a, x, b, eps : réels

Ecrire ("introduire un entier ")
Lire a

x=a
eps=0,0000000001

Faire
b=x
x=(b + a /b)/2
Ecrire (" ",x)
Jusqu'à tant que (Math.Abs(x - b) > eps)

Ecrire (" est =")
Ecrire ("", x)

Fin

CHAP II RECURSIVITE

Les modules ou sous-programme

2.1 Utilité des modules

Plus les programmes deviennent complexes, plus la mise en œuvre des algorithmes permettent de les résoudre devient fastidieuse ; il devient alors utile de décomposer le problème en sous-problèmes indépendants et de résoudre chaque sous problème de façon autonome. Un programme associé à un sous-programme est appelé un sous-programme ou module.

L'utilisation des modules offre plusieurs avantages notamment :

- La simplification de la complexité des programmes en évitant des recopiés inutiles, ce qui implique le raccourcissement du programme par l'utilisation des appels aux sous-programmes au lieu de reproduire les mêmes instructions
- La possibilité de diviser le travail de manière à ce que la réalisation de gros programmes puisse être confiée à plusieurs personnes travaillant indépendamment au profit d'un gain de temps sensible ; la facilité de localiser les erreurs ; la lisibilité des programmes.

Il existe deux types de modules à savoir, les procédures et les fonctions.

Une procédure est un sous-programme qui produit généralement un effet qui peut être :

- Un ou plusieurs résultats ;
- La modification de la valeur de certaines variables.

La déclaration d'une procédure se fera à l'aide du mot procédure suivi d'un blanc suivi du nom de la procédure et de la liste de paramètres entre parenthèses et séparés des virgules. L'appel d'une procédure constitue une instruction.

Exemple : la formule de calcul de la factorielle d'un nombre n passer en argument s'écrit :
n!=1x2x3x…xn

```
procedure facto(n :entier)
var fact : entier
fact← 1
pour i ←2 à n faire
fact←fact*i
i suivant
fin procedure

début
var n : entier
Ecrire "entrer la valeur de n:"
Lire n
Ecrire "la factorielle est :",facto(n)
Fin
```

Une fonction est un module quoi ne produit qu'une valeur qui est le type de la fonction, sa déclaration se fait par le mot fonction et d'une parenthèse contenant la liste de paramètres séparés par des

virgules. L'appel d'une fonction se fait par l'occurrence de son nom dans une expression ou une assignation.

Exemple : Définissons une fonction simple :

Le programme suivant, qui calcule la circonférence d'un cercle en fonction de son rayon, contient deux fonctions, **main**, que nous avons déjà rencontrée et ***circonférence***. La fonction ***circonférence*** prend en argument un réel **r** et retourne la valeur de la circonférence, qui est aussi un réel :

```
package exercirc;
public class ExerCirc {
static float pi = (float) Math.PI;
public static float circonference(float r)
{
return 2* pi * r;
}
public static void main(String[] args) {
// TODO code application logic here
float c = circonference (15);
System.out.println("Circonference: " + c);
return;
}
}
```

Une fois la fonction appelée, on remplace circonférence(x) par le résultat retourné par la fonction.
Bref, la fonction est une procédure ne possédant qu'un seul élément de sortie appelé paramètre.

2.2 Passage de paramètres

Les variables du programme appelant ne sont pas lisibles depuis un sous-programme. Mais si une procédure, pour s'exécuter, a besoin de la valeur d'une variable définie dans le programme appelant, on utilise pour la lui communiquer un mécanisme dit "passage de paramètres".

Un paramètre est donc une variable algorithmique servant à informer le module des variables et des valeurs à considérer lors de son exécution. Les paramètres utilisés dans la définition du module sont appelés paramètres formels, tandis que les paramètres utilisés dans un appel de sous-programme sont appelés paramètres effectifs ou paramètres réels.

Exemple : procédure résultat(x,y)

```
Début
Numérique : r
r ← x % y
Fin
```

Dans le module englobant, l'instruction résultat (23,2).
La correspondance entre paramètre formelle et paramètres effectifs est déterminée par la position :

Les paramètres effectifs 23 et 2 correspondent au paramètre formel x et y, x et y sont des paramètres donnés tandis que `"r"` est un paramètre résultat.

NB : Les fonctions que l'on a étudiées jusqu'à présent sont qualifiées d'itératives, dans le sens où le traitement qu'elles implémentent se base soit sur une unique exécution de leur code, soit sur une répétition basée sur des boucles.

Il est cependant possible de définir une autre sorte de fonctions, qui réalisent une répétition basée sur la notion de récursivité : une telle fonction s'appelle elle-même (que ce soit directement ou indirectement).

Informellement, une fonction récursive est une fonction qui s'appelle elle-même. L'intérêt d'un tel concept est qu'il correspond bien à l'idée intuitive qu'on se fait d'un procédé de calcul automatique, qui consiste à réappliquer une même procédure sur des objets similaires.

Dans les langages de programmation, cette idée peut s'exprimer à l'aide de boucles, ou encore à l'aide de fonctions récursives. En fait on verra que de nombreux algorithmes s'écrivent plus naturellement de cette dernière façon.

2.3 Fonctions numériques récursives

En mathématique, on a souvent tendance à définir les objets, et en particulier des fonctions, sans en préciser clairement une représentation, et encore moins une méthode pour en calculer la ou les valeurs. Cette situation est d'ailleurs un peu paradoxale, quand on sait les exigences de rigueur propres à cette discipline, puisqu'on peut avoir ainsi l'impression de raisonnements rigoureux fondés sur des prémisses floues. Par exemple, reprenons l'une des définitions de la factorielle qu'on trouve régulièrement dans les manuels scolaires :

n! =1 x 2 x 3 x……x n

Si l'on voit fort bien de quoi il s'agit, on peut tout de même s'interroger sur le sens exact des points de suspension : en aucun cas, cette formule ne fournit un modèle calculatoire explicite des valeurs de la fonction. Une définition alternative de la factorielle, fait intervenir un opérateur de produit :

$$\mathbf{n!} = \prod_{i=1}^{n} i$$

Il est pourtant possible de donner une définition définir non ambiguë de la factorielle par une formule de récurrence :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

L'algorithme d'une fonction factorielle :

```
Fonction Facto (n :entier)
Si n=0 alors Renvoyer 1
Sinon Renvoyer n*Facto(n-1)
Fin fonction
```

Dans le cas d'un calcul de factorielle, on s'arrête quand on arrive au nombre 1, pour lequel la factorielle est par définition 1.

On peut alors transcrire cette formule directement, c'est-à-dire en quelque sorte sans réfléchir, en une fonction récursive de n'importe quel langage de programmation, ce qui donne par exemple en C :

```
 #include <stdio.h>
 #include <stdlib.h>

int factorielle (int n){
if (n==0)
return 1;
else
return n * factorielle(n-1);
}

int main()
{
Printf ("la factorielle est %d", factorielle(4));
 return 0;
}
```

Il peut encore exister des définitions de fonctions qui, tout en n'étant pas ambiguës, ne fournissent pas de procédé explicite de calcul des valeurs. Considérons l'exemple du *plus grand commun diviseur* (PGCD) de deux entiers naturels ***n*** et ***m***. Cette définition n'indique rien sur la façon de le déterminer en pratique : on utilise l'algorithme d'Euclide qui de déduit du théorème suivant :

$$\forall (n,m) \in \mathbb{N}^2,\ n \geq m,\ \mathrm{pgcd}(n,m) = \begin{cases} n & \text{si } m = 0 \\ \mathrm{pgcd}(m, n \bmod m) & \text{sinon} \end{cases}$$

Dont la traduction en fonction récursive est immédiate.

Nota : un sous-programme est récursif s'il peut s'appeler lui-même (directement ou indirectement) ou Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.

Les fonctions récursives : une fonction récursive est une fonction qui pour fournir un résultat, s'appelle elle-même un certain nombre de fois.

2.3.1 Récursivité simple : la fonction s'appelle elle-même une seule fois au maximum, sans appeler d'autre fonction récursive.

Revenons à la fonction puissance x → x^n. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

L'algorithme correspondant s'écrit :

PUISSANCE (x, n)
Si n = 0 alors renvoyer 1
Sinon renvoyer x *PUISSANCE(x, n-1)

A titre d'exemple le problème de Syracuse est le suivant : soit ***m*** un entier plus grand que 1. On définit la suite u_n par u_0 =m et

$$u_{n+1} = \begin{cases} n \div 2 & \text{si } n \text{ est pair,} \\ 3n+1 & \text{sinon.} \end{cases}$$

(La notation n %2 désigne le quotient de la division euclidienne de n par 2). Il est conjecturé, mais non encore prouvé que pour tout ***n***, la suite converge vers 1. Pour vérifier numériquement cette conjecture, on écrit le programme Java suivant.

```
class Syracuse{
    public static void main(String[] args){
        int n = Integer.parseInt(args[0]);

        do{
            if((n % 2) == 0)
                n /= 2;
            else
                n = 3*n+1;
        } while(n > 1);
        return;
    }
}
```

2.3.2 Récursivité multiple : la fonction s'appelle elle-même plus d'une fois au maximum, sans appeler d'autre fonction récursive.

Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons C(n,p) en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

L'algorithme correspondant s'écrit :

COMBINAISON (n, p)
Si p = 0 ou p = n alors renvoyer 1
Sinon renvoyer COMBINAISON (n -1, p) + COMBINAISON (n-1, p-1)

2.3.3 Récursivité croisée : la fonction s'appelle indirectement, via une ou plusieurs autres fonctions.

La récursivité croisée peut être simple ou multiple, en fonction de l'ordre de la relation de récurrence.
Exemple : soit les deux suites suivantes :

$$u_n = \begin{cases} a_u \text{ si } n = 0 \\ 3u_{n-1} + 2v_{n-1} + b_u \text{ sinon} \end{cases} ; v_n = \begin{cases} a_v \text{ si } n = 0 \\ u_{n-1} + 4v_{n-1} + b_v \text{ sinon} \end{cases}$$

La fonction récursive correspondant à u est :

```
int suite_u(int n)
2  {  int resultat;
3    if(n==0)
```

```
4      resultat = A_U;
5    else
resultat = 3*suite_u(n-1) + 2*suite_v(n-1) + B_U;
7    return resultat;
8  }
```

La fonction récursive correspondant à v est :

```
1  int suite_v(int n)
2  {  int resultat;
3    if(n==0)
4      resultat = A_V;
5    else
6      resultat = suite_u(n-1) + 4*suite_v(n-1) + B_V;
7    return resultat;
8  }
```

Pour conclure sur la récursivité, trois remarques fondamentales.

- La programmation récursive, pour traiter certains problèmes, peut être très économique, elle permet de faire les choses correctement, en très peu de lignes de programmation.
- En revanche, elle est très dispendieuse de ressources machine. Car il faut créer autant de variable temporaire que de " tours " de fonction en attente.
- Toute fonction récursive peut également être formulée en termes itératifs ! Donc, si elles facilitent la vie du programmeur, elles ne sont pas indispensables.

2.3.4 Arbre des appels

Le concept d'arbre des appels permet de donner une représentation graphique de la récursivité, et ainsi de mieux la comprendre à travers sa visualisation :

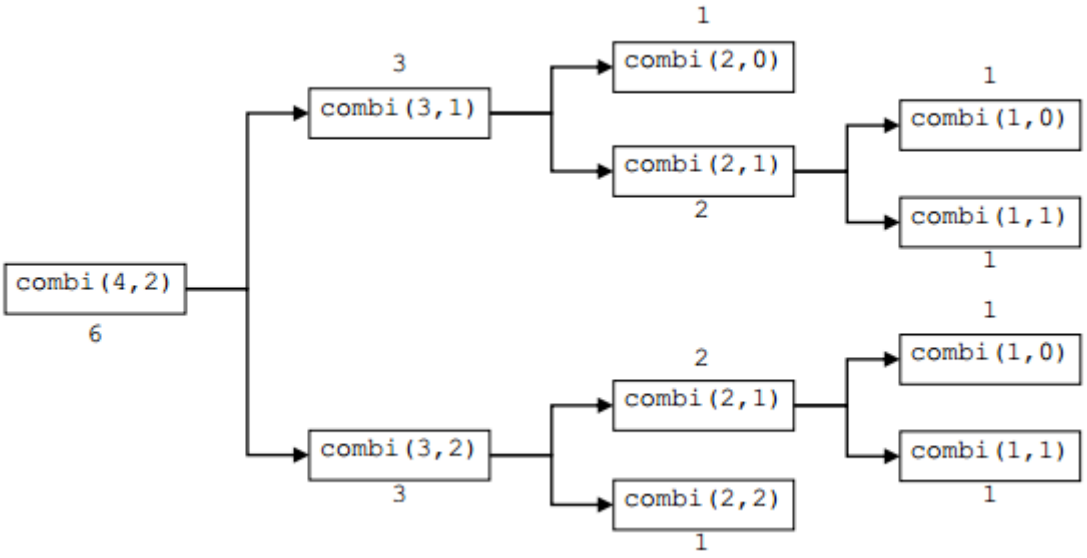
Arbre des appels: représentation graphique des appels d'une ou plusieurs fonctions s'auto ou s'inter-appelant, avec les valeurs des paramètres et les résultats renvoyés.

Par s'auto-appeler, on veut dire que la fonction s'appelle elle-même. Par s'inter-appeler, on veut dire que plusieurs fonctions s'appellent les unes les autres.

Si jamais la récursivité est simple, alors l'arbre est linéaire, c'est-à-dire qu'il ne contient qu'une seule branche. Si elle est multiple, tout dépend du nombre d'appels récursifs effectués.

Exemples :

COMBINAISON (n, p)
Si p = 0 ou p = n alors renvoyer 1
Sinon renvoyer COMBINAISON (n -1, p) + COMBINAISON (n-1, p-1)



2.4 Portée des objets dans un programme modulaire

Parmi les variables utilisées dans un programme modulaire, on distingue les variables globales et les variables locales.

Une variable est dite globale lorsqu'elle est déclarée au début du programme principal, c'est-à-dire sa valeur peut être consultée et modifiée à tous les niveaux, sauf à l'intérieur d'un sous-programme ou une autre variable est déclarée avec le même identificateur.

Une variable locale est une variable déclarée dans un module M est « visible » depuis sa déclaration jusqu'à la fin du module M, sauf à l'intérieur d'un sous-module de M où une autre variable a été déclarée avec le même identificateur. La transformation d'une variable au niveau local n'affecte pas sa valeur au niveau global.

2.5 Passage de paramètres par valeurs et par adresse (par référence ou par variable)

- Dans un passage par valeur, c'est à dire que la valeur du paramètre effectif est recopiée dans le paramètre formel correspondant. On a deux entités distinctes. Si la fonction modifie le paramètre formel, le paramètre effectif n'est lui en rien modifié.

Notre méthode :

```
Fonction notre_methode ( x : entier)
X =12
Retourner (x)
FIN
```

La méthode principale :

```
Début
Var a : entier
 a←6
Afficher ("Avant le passage de la méthode a = " + a)
Afficher ("Au passage de la méthode a = " + notre_methode(a))
Afficher ("Après le passage de la méthode a = " + a)
```

Fin

Résultats :

- Avant le passage de la méthode a=6
- Au passage de la méthode a=12
- Après le passage de la méthode a =6

- Dans un passage par référence, le paramètre effectif et le paramètre formel sont une seule et même entité. Si la fonction modifie le paramètre formel, le paramètre effectif est lui aussi modifié. Du point vu syntaxique, dans le passage par adresse le mot var est placé devant le paramètre formel et le paramètre effectif lors de la déclaration du sous-programme et dans l'appel.

Notre méthode :

```
Fonction notre_methode (var x : entier)
X=8
Retourner (x)
FIN
```

La méthode principale :

```
Début
Var a : entier
 a←2
Afficher ("Avant le passage de la méthode a = " + a)
Afficher ("Au passage de la méthode a = " + notre_methode(var a))
Afficher ("Après le passage de la méthode a = " + a)
Fin
```

Résultats :

- Avant le passage de la méthode a=2
- Au passage de la méthode a=8
- Après le passage de la méthode a =8

2.6 Comparaison des avantages et des inconvénients des 2 modes

a) Passage par adresse (référence)

Avantages : rapidité d'accès aux données, moindre occupation mémoire puisqu'il ne s'agit que d'une adresse. Inconvénients : mode dangereux à cause de non protection des données et de la nécessité qu'il y a de connaître la façon dont sont implantées physiquement les données sur la machine.

b) Passage par valeur

Avantage : sécurité et protection des informations

Inconvénients : lenteur due à la recopie des données et doublement de la place mémoire occupée (mais convient bien pour des variables simples).

Exercices :

1) Ecrire un algorithme qui demande une chaines binaires fixe à l'utilisateur celui-ci entrera ensuite, la convertit en décimal.

   Exemple : $(111,111)_2 = 7, 875\ (\ )_{10}$

2) A) Quelles sont les résultats successives prises pour chaque variable pendant l'exécution de l'algorithme ci-après, lorsque les valeurs lues sont : 20 et 8 ; 30 et 14 ;17 et 27 ?

```
Fonction récursif (p : entier, q : entier) : entier
        Si q mod p=0 alors
            Retourner p
        Sinon
            Retourner récursif (q, p mod q)
Finsi
```

   B) Que fait cet algorithme et donner son arbre d'appel.

3) Ecrivez un algorithme d'une fonction récursive qui demande une chaîne à l'utilisateur et qui affiche à l'écran son inverse, le point d'arrêt : si la chaine est vide, plus d'appel récursif, on retourne une chaine vide.

4) On donne une fonction F(m,n) avec m > 0 et n > 0, définie récursivement par

$$F(m,n)=\begin{cases} 0 & si\ \ m < n \\ F(m-n,n)+1 & sinon \end{cases}$$

   a) Evaluer F (3,4), F (15,4) et F (58,7)
   b) Quelle est l'objet de cette fonction et donner son arbre d'appel ?

5) Ecrire un algorithme qui demande un entier fixe à l'utilisateur celui-ci entrera ensuite, la convertit en binaire.

Exemple : $(15)_{10} = 1111\ (\ )_2$

6) On donne une fonction F (a, b, c) =F2 (F1 (a, b), c), ∀ a, b, c ≥ 0, définie récursivement par :

$$F1\ (a, b) = \begin{cases} a & si\ \ b\ \%\ a = 0 \\ F(\ b,\ a\ \%\ b) & sinon \end{cases}$$

$$= \begin{cases} F1(a,b) & si\ \ c\ \%\ F1(a,b) = 0 \\ F2(\ c,\ F1(a,b)\ \%\ c) & sinon \end{cases} \quad F2\ (F1\ (a, b), c)$$

A) Evaluer F2 si a = 30, b = 18 et c = 317 ; F2 si a=30, b=14 et c = 28
B) Quelle est l'objet de cette fonction et donner son arbre d'appel?

7) Quelles sont les résultats successives prises pour chaque variable pendant l'exécution de l'algorithme ci-après, lorsque les valeurs lues sont :david,algo et abc?

```
Fonction inv( ch :chaines ) : chaines
Si ch= "" alors
Inv= ""
Sinon
Inv=ch.substring( len(ch)-1,1)+ inv(ch.substring(0,len(ch)-1))
Finsi
Fin Fonction
```

CHAP III LES STRUCTURES DE DONNEES

En informatique une structure de données est une structure logique destinée à contenir des données afin de leur donner une certaine organisation qui facilite leur traitement.

Nous allons dans ce chapitre en étudier quelques exemples élémentaires.

Les structures des données sont soit linéaires, soit non linéaires. Une structure de données est dite linéaires quand ses éléments forment une séquence ou, en d'autres termes, une liste linéaire. Il existe deux manières fondamentales de représenter en mémoire de telles structures linéaires :

- Représenter la relation linéaire entre les éléments au moyen position mémoires séquentielles. Ces structures linéaires sont appelées *tableaux*
- Représenter la relation linéaire entre les éléments au moyen de pointeurs ou liens. Ces structures linéaires sont appelées *listes chaînées*.
- Lorsque les données à stocker forment un univers U trop vaste, les structures de base ne font plus l'affaire. Il faut trouver un compromis entre le temps d'exécution et l'espace mémoire informatique (complexité spatiale) : la table de hachage.

Et les structures non linéaires sont les arbres et les graphes.

Les choix d'une structure dans une situation donnée dépendent de fréquence relative avec laquelle l'utilisateur effectue ces différentes opérations sur la structure particulière choisie.

Définition

Un tableau (array en anglais) est une liste à taille fixe qui est indexé c'est-à-dire l'accès à chaque élément du tableau se fait moyennant l'indice , et qui détermine l'emplacement de l'élément.

III.1 Les tableaux linéaires

Si par exemple, T est un tableau de 7 variables, alors chacune d'elles sera numéroté et il sera possible de la retrouver en utilisant simultanément le nom du tableau et l'indice de la variable. Les différentes variables de T porteront des numéros de 1 à 7, et nous appellerons chacune de ces variables un élément de T. voici comment ils seront rangés dans la mémoire.

Adresse de la case mémoire	Valeur stockée
1704	345
1705	0
1706	-25
1707	7
1708	42
1709	23
1710	1337

Soit Loc(TA[K])=adresse de l'élément TA[K] du tableau TA.
Les éléments de TA sont stockés dans des cases mémoires consécutives. Il est donc inutile que l'ordinateur garde la trace de chaque élément de TA, il suffit de garder celle de son premier élément, Base(TA), appelée adresse de base de TA.

En utilisant cette adresse Base(TA), l'ordinateur calcule l'adresse de n'importe quel élément de TA par la formule suivante :

Loc(TA[k])=base (TA)+w(k-limite inférieure)

Où w est le nombre de mots par case mémoire pour le tableau TA. Le temps nécessaire au calcul Loc(TA[k]) est particulièrement le même pour tout k.
On peut remarquer qu'étant donné tout indice k, il est possible de localiser et d'atteindre le contenue de TA[k] sans avoir à échantillonner en autre élément de TA.

III.1.1 Déclaration

Comme les variables d'un tableau doivent être de même type, il convient de préciser ce type au moment de la déclaration du tableau. De même, on précise lors de la déclaration du tableau le nombre de variables qu'il contient. La syntaxe est :

<var > <nom> (<taille>) : type

Par exemple : numérique : T (6)
Déclare un tableau T contenant 6 variables de type numérique.

III.1.2 Accès aux éléments

On peut accéder directement à un élément d'un vecteur à l'aide de son indice. Accès direct s'oppose à l'accès séquentiel qui consiste, pour accéder à un élément, à parcourir tous les éléments qui le précèdent.

Les éléments d'un tableau à n éléments sont indicés de 1 à n. On note T[i] l'élément d'indice i du tableau T. Les sept éléments du tableau de l'exemple ci-avant sont donc notés T(1), T(2), T(3),…. et T[7].

III.1.3 Traitement courants sur les vecteurs

Les traitements courants sur les vecteurs sont :

- Le parcours d'un tableau ;
- La recherche d'une information dans un vecteur ;
- La fusion de vecteurs ;
- La mise en jour des informations d'un vecteur (ajout, suppression, modification) ;
- Le tri des informations d'un vecteur

III.1.4 Parcours d'un vecteur

Problème : sur chaque élément E[i] d'un vecteur E, effectuer un traitement T(e(i)), ce type de boucle s'appelle un parcours de tableau. En règle générale on utilise des boucles pour manier les tableaux, celles-ci permettent d'effectuer un traitement sur chaque élément d'un tableau. La déclaration ci-dessus est celle d'un tableau de 10 éléments de type numérique appelé E. Il convient ensuite d'effectuer les saisies des 10 valeurs. On peut par exemple procéder de la sorte :

Pour i allant de 0 à 9
Saisir E[i]
fin pour

Ensuite, il faut saisir une valeur à rechercher dans le tableau :

III.1.5 Recherche d'une information dans un vecteur

La recherche d'une information dans un vecteur est un problème fréquent en informatique, il s'agit de déterminer si une information donnée est ou non présente dans un vecteur.

Objet chercher(object x){ return T[x] ; }

III.1.5.1 Recherche séquentielle

Principe : la recherche séquentielle consiste à examiner les éléments du vecteur un à un en commençant par le premier jusqu'à ce que l'on rencontre l'information cherchée ou l'on atteigne le dernier élément.

$$C(n)=\sum_{i=1}^{n} n_i\, p_i$$

Où p_i : représente la probabilité d'observer i (parmi les instances de taille n) et Soit C(n) le nombre de fois que l'opération de base (choisie) est effectuée par l'algorithme sur une instance de taille n.

III.1.5.1.1 Recherche séquentielle dans un tableau linéaire (une dimension) non trié.

Spécification de l'algorithme

- Soit t un tableau d'entiers de 1…n éléments non rangés
- On recherche le rang (la place) de l'élément Elt dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément Elt n'est pas présent dans le tableau t).

Version tantque avec ''et alors '' (opérateur et optimisé)
i← 1
tantque (i≤n) et alors (t(i) ≠ Elt) faire
i ← i+1
fin tantque
si i ≤n alors
rang ← i
sinon rang ← -1
finsi

Version tantque avec '' et '' (opérateur et non optimisé)
i ← 1
tantque (i< n) et (t(i)<> Elt) faire
i←i+1
fin tantque
si t(i)=Elt alors
rang ← i
sinon
rang← -1
finsi

Version pour avec instruction de Sorti (sortirsi)

Pour i ← 1 jusqu'à n faire

Sortisi t(i)=Elt
Fin pour
Si i≤ n alors
Rang ← i
Sinon
Rang ← -1
Finsi

III.1.5.1.2 Recherche séquentielle dans un tableau linéaire (une dimension) déjà trié

Objectif d'écrire un algorithme effectuant une recherche séquentielle dans un tableau linéaire trié avant la recherche.

- Soit t un tableau d'entiers de 1…n éléments rangés par ordre croissant par exemple
- On recherche le rang (la place) *d* l'élément *Elt* dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément Elt n'est pas présent dans le tableau t).

On peut reprendre sans changement les algorithmes précédents travaillant sur un tableau non trié.
On peut aussi utiliser le fait que le dernier élément du tableau est le plus grand élément et s'en servir comme une sorte de sentinelle. Ci-dessous deux versions utilisant cette dernière remarque.

Version Tantque

Si t(n) ≥ Elt alors
i←1
tantque t(i) < Elt faire
i ← i+1
fin tantque
si t(i)=Elt alors
Renvoyer rang ← i // retour du résultat et sortie du programme
Finsi
Finsi
Renvoyer rang ← -1 // retour du résultat et sortie du programme
Version Pour :

Si t(n) ≥ Elt alors
Rang ← -1
Pour i ← 1 jusqu'à n-1 faire
Sortisi t(i) ≥ Elt // sortie de boucle
Fin pour
Si t(i) = Elt alors
Renvoyer rang ← i // retour du résultat et sortie du programme
Finsi
Finsi
Renvoyer rang ← -1 // retour du résultat t sortie du programme

Un programme relatif à la recherche linéaire :
La manière la plus simple de ranger une grande quantité d'information est de la mettre dans un tableau, qu'on aura à parcourir à chaque fois que l'on cherche une information. Considérons le petit dictionnaire contenu dans la variable dico du programme ci- dessous :

```
package dico;
public class Dico {

   public static void main(String[] args) {

 String[ ] dico = {"maison", "bonjour", "moto", "voiture","artichaut", "Palaiseau"};

       boolean estdans = false;
       for(int i = 0; i < dico.length; i++)
         if("david".compareTo(dico[i]) == 0)
              estdans = true;
     if(estdans){
        System.out.println("Le mot est présent");
     }else{
         System.out.println("Le mot n'est pas présent");
     }
   }
}
```

III.1.5.2 Recherche dichotomique (binaire)

Au lieu de rechercher séquentiellement du premier jusqu'au dernier, on compare l'élément Elt à chercher au contenu du milieu du tableau. Si c'est le même, on retourne le rang du milieu, sinon l'on recommence sur la première moitié (ou la deuxième) si l'élément recherché est plus petit (ou plus grand) que le contenu du milieu du tableau.

Spécifications de l'algorithme :

- Soit t un tableau d'entiers de 1…n éléments triés par ordre croissant.
- On recherche le rang (la place) de l'élément *Elt* dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément *Elt* n'est pas présent dans le tableau t).

$$C(n)=log_2^n$$

Version itérative :

```
Algorithme : Recherche dichotomique
    Variables : bas, milieu, haut, rang : entiers
     bas←1
     haut←n (n la taille du vecteur)
     rang←-1
              Répéter
         milieu←(bas+haut) div 2
   si elt= t[milieu] alors
                    rang←milieu
   sinon si t[milieu] < elt alors
                    bas←milieu+1
           sinon
                    haut←milieu-1
           finsi
   finsi
```

Jusqu'à (elt=t[milieu] ou (bas > haut))

Un programme de la recherche binaire en langage JAVA :

```
package dico;
public class Dico {

static boolean dichoRec(String[] dico, String mot, int g, int d){
int m, cmp;
if(g >= d) // l'intervalle est vide
return false;
m = (g+d)/2;
cmp = mot.compareTo(dico[m]);
if(cmp == 0)
return true;
else if(cmp < 0)
return dichoRec(dico, mot, g, m);
else
return dichoRec(dico, mot, m+1, d);
}

static boolean estDansDico(String[ ] dico, String mot){
return dichoRec(dico, mot, 0, dico.length);
}

public static void main(String[ ] args) {
       // TODO code application logic here
       String[] dico = {"Palaiseau", "artichaut","bonjour", "maison","moto", "voiture"};

   for(int i = 0; i < dico.length; i++){
     System.out.print("Le mot '"+ dico[i]);
         if(estDansDico(dico,dico[i]))
     System.out.println("est dans le dictionnaire");
         else
     System.out.println("n'est pas dans le dictionnaire");
   }

   }
}
```

III.1.6 Fusion

Soient A et B deux vecteurs respectivement m et n éléments. Nous souhaitons les fusionner en un vecteur unique C. Nous pouvons ce faire mettre arbitrairement A et B dans C.

Exemple d'un algorithme de fusion

Soient trois tableaux T(i), S(j), F(k) indicés par i, j et k. nous voulons effectuer la fusion de deux tableaux s(j) et T(i) supposés non vides dans F(k). Sachant que T est de taille n et S de taille m.

```
Algorithme
        Début
        Lire T
        Lire S
i ← 1
            k ←1
      10 F(k) ← T(i)
            i←i+1
             k←i
     si i ≤ n aller à 10
 sinon j← 1
   20        k←j+n
            F(k) ←S(j)
j←j+1
          si j ≤ m alors aller à 20
          sinon afficher F
       Fin
```

III.1.7 Mise à jour

Nous parlerons ici de l'ajout et de la suppression d'un élément dans un vecteur. Supposons que nous voulions insérer un élément d dans un vecteur A de taille n. Si les éléments de A ne sont pas ordonnés, il n'y a qu'à ajouter d à la fin du vecteur. Si B est le vecteur mis à jour déclaré avec une taille supérieure à celle de A, la mise à jour se fait par les actions suivantes :

```
Pour i de 1 à n
B(i)←A(i)
B(i+1) ←d
```

Ce pendant si le vecteur A est trié, il faut d'abord trouver la position p où d doit être inséré avant de la faire. C'est de nouveau un travail de recherche. Une fois p trouvé, on ne peut pas affecter d à A(p) de peur d'écraser A(p), par conséquent, il faut d'abord libérer la place en déplaçant la partie du vecteur A(p), A(p+1),…, A(n) d'une position vers la droite. Appelons B le vecteur mis à jour, la taille de B sera taille (A)+1 en cas d'ajout et taille (A)-1 en cas de suppression.

L'ajout correspond à l'action :

```
Void ajout (Object x, Object valeur) {
T[x]=valeur ;
}
```

```
Pour i allant de n à p
Faire B(i+1) ← A(i)
Fin faire
B(p) ← d
```

Pour terminer, nous incrémentons d'une unité le nombre N, afin de tenir compte de l'adjonction du nouvel élément du tableau TA.

1. [Initialisation du compteur] poser j=n
2. Répéter étape 3 et 4 tant que j ≥ k
3. [Déplacer l'élément j vers la queue] poser TA[j+1] = TA[j]
4. [Décrémenter le compteur] poser j=j-1
   [Fin de boucle de l'étape 2]
5. [Insérer l'élément] poser TA[k]=IT
6. [Réinitialiser N] poser n=n+1
7. Exit

Pour supprimer un élément A(p), il suffit de déplacer les éléments A(p+1)…A(n) d'un rang vers la gauche si A(p) n'est pas le dernier élément du vecteur , ce quoi correspond à l'action :

Pour i de p à n-1 faire
B(i) ← A(i+1)
Fin faire
Si p=n il suffit de mettre les éléments A(1)…A(n-1) dans B c'est-à-dire à d
Pour i de 1 à n-1
B(i) ← A(i)

L'algorithme suivant supprime l'élément d'ordre k d'un tableau linéaire TA et l'affecte à une variable IT. TA est un tableau linéaire à N éléments et k est tel que k≤n

1. Poser IT=TA[k]
2. Répéter pour j = k à n-1
   [Déplacer l'élément d'ordre j+1 vers la tête]
   Poser TA[j]=TA[j+1]
   [Fin de boucle]
3. [Réinitialiser le nombre n d'élément de TA]
   Poser n=n-1
4. Exit

Programme de suppression d'un élément dans un vecteur :

```
Void supprimer (Object x) {
Object y = T[x] ;
T[x]=null ;
Return y ;
}
```

III.1.8 Tri d'un vecteur

Soit V[1…n] un vecteur dans un élément E muni d'une relation d'ordre <. Trier v consiste à construire un vecteur v' tel que l'application v' soit monotone c'est-à-dire trier V désigne l'opération de reclassement des éléments de V de manière qu'il soit dans un ordre croissant. Nous présentons dans les lignes qui suivent trois algorithmes élémentaires de tri.

III.1.8.1 tri par méthode des bulles

Le principe de ce tri consiste à imaginer que les éléments les plus petits ou les plus légers partent comme des bulles à la surface. On effectue des passages successifs sur le vecteur de bas en haut. A chaque étape, si deux éléments sont en ordre inverse. La conséquence de cette opération est qu'à

l'issue du premier passage, l'élément le plus léger remonte jusqu'à la première position (à la surface). Au second passage, la deuxième plus petite valeur remonte à la deuxième position et ainsi de suite. Il est inutile au deuxième passage d'essayer de faire remonter l'élément jusqu'à la première position puisqu'on est sûr que cette position est définitivement qu'aux éléments situés entre le bas et le premier passage, on ne s'intéresse qu'aux éléments situés entre le bas et la première position.

$$C(n)=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}=o(n^2)$$

En d'autres termes, le temps nécessaire à l'exécution de l'algorithme de tri bulle est propositionnel à n^2, où n est le nombre d'items entrants.

Nous pouvons de ce fait proposer l'algorithme suivant : ici l'indice de tableau commence par 1

Spécifications de l'algorithme

Algorithme : tri-a-bulles
Var i,j,n,temp : entiers
Var entrée-sortie tab : entiers de 1 à n éléments
Début

```
Pour i de n jusqu'à 1 faire              // recommence une sous-suite (a1,a2,…ai)
    Pour j de 2 jusqu'à i faire          // échange des couples non classés de la suite
        Si tab(j-1) > tab(j) alors       // aj-1 et aj non ordonnés
Temp ← tab(j-1)
      tab(j-1) ←  tab(j)
tab(j) ← temp                            // on échange les positions de aj-1 et aj
finsi
fin pour
fin pour
fin
```

Dans le cas où l'on démarre le tableau à l'indice zéro, on change les bornes des indices i et j

```
Pour i de n jusqu'à 0 faire
Pour j de 1 jusqu'à i faire
< Instructions identiques >
Fin pour
Fin pour
```

III.1.8.2 tri par insertion

La méthode consiste pour tout i compris entre 2 et n, à insérer tab(i) parmi les i-1 premiers éléments de la liste déjà rangés.

Plan de l'algorithme niveau 1

Lecture du nombre n d'éléments dans la liste (taille de la liste)
Lecture de la liste
Pour i de 2 à n répéter
Recherche dans la liste des i-1 premiers éléments, le rang du premier élément qui se trouve après l'élément que l'on veut ranger
Insérer cet élément parmi les i-1 premiers éléments de la liste

Ecrire la liste
On utilise une mémoire v de type entier qui contiendra le rang où l'on doit insérer l'élément que l'on veut ranger dans la partie de la liste déjà rangée.

Algorithme

```
Pour i de 2 jusqu'à n faire        //la partie non encore triée (ai,ai+1,…,an)
  v ← tab(i)                       // l'élément frontière :ai
     j← i                          // le rang de l'élément frontière
tantque tab(j-1) > v faire         // on travaille sur la partie déjà triée (a1,a2,…,ai)
tab(j) ← tab(j-1)                  // on décale l'élément
       j ← j-1                     // on passe au rang précédent
fin tantque
    tab(j) ← v                     // on recopie ai dans la place libérée
      fin pour
fin
```

Un programme en JAVA :

```
package TriInsertion;
public class TriInsertion {

  static int[] triInsertion(int[] t){
int n = t.length, j, tmp;
for(int i = 1; i < n; i++){
                    j = i;
// recherche la place de t[i] dans t [0..i-1]
while((j > 0) && (t[j-1] > t[i]))
j--;
// si j = 0, alors t[i] < = t[0]
// si j > 0, alors t[j] > t[i] >= t[j-1]
// dans tous les cas, on pousse t[j..i-1] vers la droite
tmp = t[i];
for(int k = i; k > j; k--)
t[k] = t[k-1];
t[j] = tmp;
}
return t;
 }
```

Second niveau d'analyse

On va construire des procédures pour résoudre les problèmes suivants :
Calculer le rang d'insertion du $p^{ième}$ élément du tableau dans le tableau constitué de p-1 premiers éléments du tableau, soit la procédure rang. Intercaler le $p^{ième}$ élément du tableau à la $k^{ième}$ position soit la procédure coincer

Analyse de la procédure Rang

Les paramètres de la procédure sont les suivants : paramètres données : n la taille du tableau, liste le tableau, p le rang de l'élément que l'on veut insérer ; paramètre résultat : r le rang où l'on veut ranger liste (p) dans liste des p-1 premiers éléments. Puisque la procédure ne comporte qu'un seul paramètre résultat, elle sera une fonction. Nous allons utiliser la méthode dichotomique.

Nous obtenons la fonction suivante :

```
Fonction Rang(n,liste,p) :entier
Paramètres données : n,p :entier,liste :tab(1…100) d'entiers
nbre ← liste(p)
pos1←1
pos2← p-1
répéter milieu ←((pos2-pos1) div 2)+pos1)
si nbre < liste (milieu) alors
pos2← milieu
sinon pos ← milieu+1
jusqu'à pos1= pos2
si nbre < liste(pos1) alors
rang← pos1
sinon rang ← pos1+1
retour
```

Nous avons mis le contenu de liste (p) dans une mémoire nbre de manière à gagner du temps à l'exécution.

Analyse de la procédure coincer

Les paramètres de la procédure sont les suivants :

- Paramètres données : p, le rang du nombre que l'on veut intercaler ; k, le numéro de la place où l'on veut intercaler ; n, la taille du tableau.
- Paramètre donnée -résultat : la liste, tableau. Liste est un paramètre donnée -résultat car liste est modifié au cours de la procédure.

```
Procédure coincer(p, k ,n, liste)
Paramètres données :p,k,n :entiers
Paramètre donnée-résultat : liste : tab (1…100) d'entiers
Var j,z : entier
z←liste(p)
pour j de p-1 à k répéter
liste(j+1) ← liste(j)
liste(k) ← z
Retour
```

On peut maintenant écrire le programme principal appelant la fonction Rang et la procédure Coincer.

```
Programme principal
Var j,k,n : entier ;liste :tab(1…100) d'entiers
Lire n
Lire liste
Répéter

Appel Coincer (j,Rang(n,liste,j),n,liste)
```

j← j+1
Jusqu'à j>n
Pour k de 1 à n
Faire
Ecrire liste (k)
Fin programme principal.

III.1.8.3 tri par sélection

L'algorithme de tri sélection appliqué à un vecteur opère de la manière suivante :
Il trouve d'abord l'élément le plus petit de la liste et le place en premier position, puis il trouve l'élément suivant et le place en seconde position, et ainsi de suite.

L'algorithme :

J=0
Pour k=1 à n-1 faire
J=j+1
min=A(j)
d=j
Pour i =j+1 à n faire
Si min > A(i) alors
min=A(i)
d=i
Finsi
i suivant
T=A(k)
A(k)=A(d)
A(d)=T
K suivant

```
package tri_Selection;

public class Tri_Selection {

 static int[] triSelection(int[] t){

    int n = t.length, m, tmp;

    for(int i = 0; i < n-1; i++){

          m = i;

     for(int j = i+1; j < n; j++)

          if(t[j] < t[m])

             m = j;  tmp = t[i];  t[i] = t[m]; t[m] =
tmp;

                    }

              return t;

}  }
```

```
 public static void main(String[] args)
{

      int[] t = {3, 5, 7, 3, 4, 6};

        t = triSelection(t);

           return;

   }
```

III.1.8.4 Un tri rapide : le tri par fusion

Il existe plusieurs algorithmes dont la complexité atteint O (n log n) opérations, avec des constantes et des propriétés différentes. Nous avons choisi ici de présenter uniquement le tri par fusion.
Ce tri est assez simple à imaginer et il est un exemple classique de diviser pour résoudre. Pour trier un tableau, on le coupe en deux, on trie chacune des deux moitiés, puis on interclasse les deux tableaux.
On peut déjà écrire simplement la fonction implantant cet algorithme :

```
static int[] triFusion( int[] t){

if(t.length == 1) return t;
int m = t.length / 2;
int[ ] tg = sousTableau(t, 0, m);
int[ ] td = sousTableau(t, m, t.length);

// on trie les deux moities
tg = triFusion(tg);
td = triFusion(td);
// on fusionne
```

```
return fusionner(tg, td);
}
```

En y ajoutant la fonction qui fabrique un sous-tableau à partir d'un tableau :

```
// on crée un tableau contenant t[g..d[

static int[ ] sousTableau( int[] t, int g, int d){
int[ ] s = new int[d-g];
for(int i = g; i < d; i++)
s[i-g] = t[i];
return s; }
```

Il nous reste à expliquer comment on fusionne deux tableaux triés dans l'ordre.
Pour programmer cette fusion, on va utiliser deux indices g et d qui vont parcourir les deux tableaux tg et td. On doit également vérifier que l'on ne sort pas des tableaux.

Cela conduit au code java suivant :

```
static int[ ] fusionner(int[ ] tg, int[ ] td){
int[ ] f = new int[tg.length + td.length];
int g = 0, d = 0;
for(int k = 0; k < f.length; k++){
// f[k] est la case à remplir
if(g >= tg.length) // g est invalide
f[k] = td[d++];
else if(d >= td.length) // d est invalide
f[k] = tg[g++];
else
// g et d sont valides
if(tg[g] <= td[d])
f[k] = tg[g++];
else // tg[g] > td[d]
f[k] = td[d++];
}
return f;
}
```

D'autre part, il existe une version non récursive de l'algorithme de tri par fusion qui consiste à trier d'abord des paires d'éléments, puis des quadruplets, etc. Nous laissons cela à titre d'exercice.

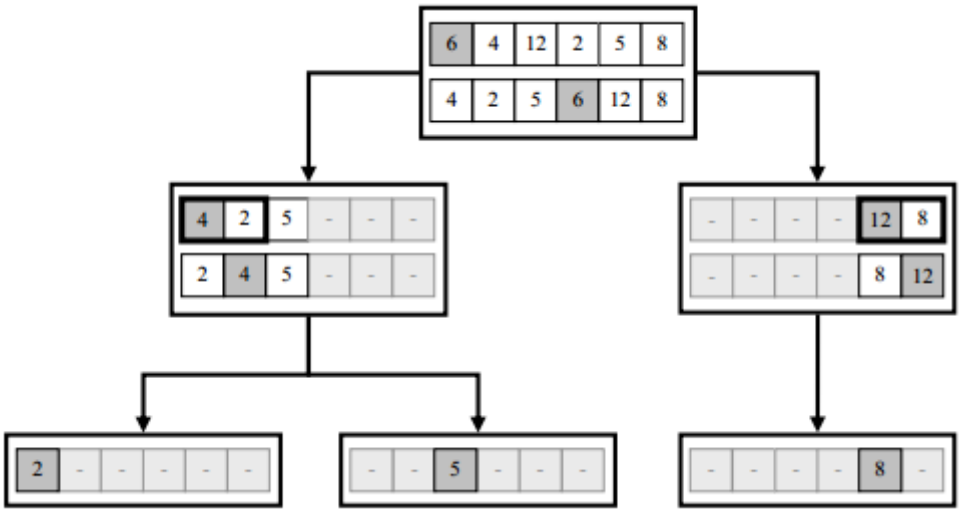
III.1.8.5 Tri rapide ou tri par segmentation

Principe
Le tri rapide est également appelé : tri de Hoare, tri par segmentation, tri des bijoutiers…

L'algorithme est le suivant :
1. On choisit un élément du tableau qui servira de pivot.
2. On effectue des échanges de valeurs dans le tableau, de manière à ce que :
   - Les valeurs inférieures au pivot soient placées à sa gauche.
   - Les valeurs supérieures au pivot soient placées à sa droite.

3. On applique récursivement ce traitement sur les deux parties du tableau (i.e. à gauche et à droite du pivot).

Remarque : on peut remarquer que le choix du pivot est essentiel. L'idéal est un pivot qui partage le tableau en deux parties de tailles égales. Mais déterminer un tel pivot coûte trop de temps, c'est pourquoi il est en général choisi de façon arbitraire. Ici par exemple, on prend la première case du tableau. Entre ces deux extrêmes, des méthodes d'approximation peu coûteuses existent pour faire un choix de compromis.



Dans figure ci-dessus, la valeur indiquée en gris correspond au pivot sélectionné pour diviser le tableau en deux.

Implémentation

Soit void tri_rapide(int tab[ ], int d, int f) la fonction récursive qui trie le tableau d'entiers tab de taille N en utilisant le principe du tri rapide.

Les paramètres d et f marquent respectivement le début et la fin du sous-tableau en cours de traitement. Ces deux paramètres valent donc respectivement 0 et N-1 lors du premier appel.

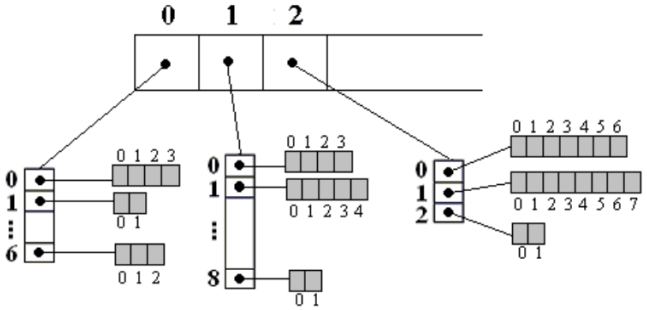
```
void tri_rapide(int tab[ ], int d, int f)
{     int  taille=f-d+1;
      Int  i, j, temp;
 if (taille>1)
     {       j=d;
       for (i=d+1;i<=f;i++)
        {  if (tab[i]<tab[d])
              {  j++;
            temp=tab[ j];
            tab[ j]=tab[ i];
            tab[ i]=temp;
             }
        }
           temp=tab[d];
           tab[d]=tab[ j];
           tab[ j]=temp;
```

```
if (j>d)
 tri_rapide(tab,d,j-1);
 if (j<f)
 tri_rapide(tab,j+1,f);
   }
}
```

III.2 Tableau multidimensionnels

III.2.1 Généralités

Les éléments des vecteurs ou tableaux linéaires dont nous avons parlé jusqu'à présent ont toujours un indice. Il y a lieu de définir des tableaux dont les éléments sont identifiés à l'aide de 2, 3,…, indices, ce sont des tableaux multidimensionnels (tableaux de déchiquetés ou en escalier).
Le schéma ci-dessous montre un tableau T à trois dimensions en escalier :



Ce schéma montre bien qu'un tel tableau T est constitué de tableaux unidimensionnels, les tableaux composés de cases blanches contiennent des pointeurs (références). Chaque case blanche est une référence vers un autre tableau unidimensionnel, seules les cases grisées contiennent les informations utiles de la structure de données et le tableau sera rangé en mémoire en une séquence de cases mémoires. De même, le langage de programmation va ranger le tableau T selon l'un des deux ordres de priorité, ligne ou colonne.

III.2.2 Principe

Par ordre de ligne priorité, les éléments sont listés de manière que leurs indices varient comme les chiffres d'un compteur kilométriques d'automobile, c'est-à-dire que le dernier indice varie en premier (le plus rapidement), le suivant varie moins rapidement ainsi de suite.

Par ordre de colonne prioritaire, les éléments sont listés de manière telle que le premier indice en premier le plus rapidement, le suivant en second et ainsi de suite.

a) Par ordre de colonne prioritaire

$$Loc[C(k_1,k_2,\ldots,k_n)]=base(C) +w[(((\ldots(E_nL_{n-1}+E_{n-1})L_{n-2})+\ldots+E_3)L_2+E_2)L_1+E_1]$$

b) Par ordre de ligne prioritaire

$$Loc[C(k_1,k_2,\ldots,k_n)]=base(c) + w\ [(\ldots((E_1L_2+E_2)L_3+E_3)L_4+\ldots+E_{n-1})L_n+E_n]$$

Avec Base (c) l'adresse du premier élément de c et w le nombre de mots par case mémoire.

III.2.4 Représentation mémoire

Soit A un tableau à trois dimensions A (2,4,3), donc A comporte 2*3*4=24 éléments. En général, les trois indices sont appelés respectivement ligne, colonne et page.

Les tableaux bidimensionnels correspondent au concept mathématique de matrice rectangulaire. Un tableau bidimensionnel m x n peut être défini comme étant un ensemble de m x n cases mémoires déterminées chacune par une paire d'entiers k,l avec $1 \leq k \leq m$ et $1 \leq l \leq n$,k et l sont appelés les indices . Le 1er indice de k indique la ligne tandis que le second *l* indique la colonne de l'élément. Chaque élément est désigné par le nom du tableau suivi d'une paire d'indices séparés par une virgule et mis entre parenthèses. Un tableau rectangulaire est représenté à l'intérieur de la machine comme une suite de m x n cases mémoires. On définit de la même manière un tableau tridimensionnel etc.

Exemple : A(m,n)=tableau a(k,l)=élément de A

Ici nous allons déclarer un tableau à 2 dimensions. Pour cela il faut rajouter une virgule entre les parenthèses indiquant que c'est un tableau à 2 dimensions (2 virgules pour un tableau à 3 dimensions, 3 virgules pour un tableau à 4 dimension etc.).
Pour parcourir un tableau a deux dimensions, nous allons appliquer la même méthode que pour un tableau linéaire, sauf que nous allons imbriquer les boucles pour afin de balayer toutes les dimensions, ici en occurrence 2.

III.3 Les structures de données dynamiques

Jusqu'ici, on a étudié des structures de données statiques, c'est-à-dire qui ont un nombre fixe d'informations, utilisées pour la représentation de données dont la taille est connue à l'avance et n'évolue pas dans le temps.

Dans la pratique, il arrive souvent que l'on ait besoin de représenter des objets dont on ne connaît pas à priori la taille, ou dont la taille est variable selon les cas ou au cours du temps.

On utilise dans ce cas des structures de données dynamiques qui peuvent évoluer pour s'adapter à la représentation de ces objets.

Exemple : On doit lire dans un fichier une suite de nombres sur lesquels on doit effectuer un traitement ultérieur. On ne connaît pas la quantité de nombres à lire et on ne veut pas les compter avant.

La première solution consiste à utiliser un tableau sur dimensionner, au risque qu'il soit trop petit ou réellement trop grand, ce qui induit une occupation mémoire inutile.

L'autre solution consiste à utiliser une structure dynamique qui s'agrandit au fur et à mesure de la lecture des nombres.

III.3.1 Structure autoréférentielle ou dynamique

- Si une structure contient des données et un pointeur vers la structure suivante, on parle de **liste chaînée**.
- Lorsque la structure contient des données, un pointeur vers la structure suivante et un pointeur vers la structure précédente, on parle de **liste chaînée double**.

- Lorsque la structure contient des données, un pointeur vers une première structure suivante et un pointeur vers une seconde, on parle d'**arbre binaire**.
- **Table de Hachage**.

Présentation

Une liste correspond à un type de données abstrait, dans le sens où il peut être implémenté de différentes façons. Nous reviendrons bientôt sur cette notion de type abstrait.

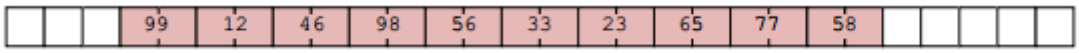
Liste : séquence d'éléments uniformes.

Ici, uniforme signifie que tous les éléments constituant la liste sont du même type. On peut donc avoir une liste d'entiers, ou une liste de réel, etc. On distingue différentes parties dans une liste :

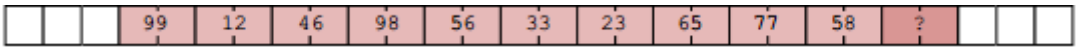
Tête de liste : le tout premier élément de la séquence.

Queue de liste : la sous-liste constituée de tous les éléments sauf le premier.

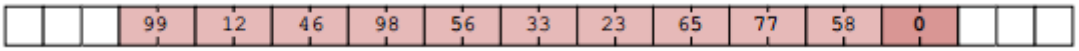
Exemple : supposons qu'on ait le tableau suivant, et que l'on veut y insérer une nouvelle valeur 0.



Alors il va falloir d'abord l'agrandir d'un élément :



Puis enfin faire l'insertion :



Comme on l'a vu quand nous avons étudié l'allocation dynamique, s'il n'est pas possible d'agrandir la zone mémoire actuellement allouée, il est possible qu'une zone entièrement différente soit réservée, ce qui implique de recopier l'intégralité du tableau ailleurs en mémoire.

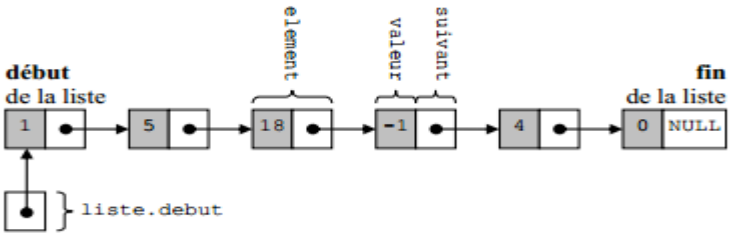
Ces différents inconvénients n'empêchent pas d'utiliser les tableaux pour représenter des listes, mais ils rendent cette approche inefficace. Ceci est dû au fait que les tableaux ont besoin de stocker leurs éléments de façon contigüe.

Pour éviter ces problèmes, il faut donc adopter une implémentation dans laquelle la place d'un élément en mémoire n'est pas liée à sa place dans la liste : c'est le but des listes chaînées.

Nous allons distinguer deux sortes de listes en fonction du nombre de lien par élément, et de leur direction : listes simplement ou doublement chaînées.

III.3.1.1 Les Listes simplement chaînées

Une liste chaînée est une structure de données dans laquelle les éléments sont rangés linéairement. Chaque élément est lié à son successeur, il est donc impossible d'accéder directement à un élément quelconque de la liste.



En pratique, nous utiliserons un pointeur pour indiquer l'adresse de l'élément suivant dans la liste. Donc, pour l'élément n de la liste, on va associer un pointeur sur l'élément n + 1.

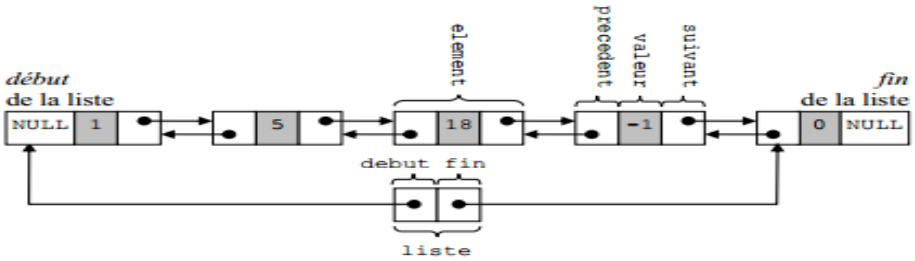
Notez que cette implémentation permet de parcourir la liste du début (premier élément) vers la fin (dernier élément). Par contre, on ne peut pas remonter la liste, i.e. passer d'un élément n à un élément $n - 1$.

Liste vide : liste qui existe en tant que variable, mais qui ne contient aucun élément (null).

III.3.1.2 Les liste doublement chaînées

Une liste doublement chaînée est une liste chaînée, à ceci près que l'on peut accéder à son prédécesseur.

Exemple : représentation graphique d'une liste doublement chaînée :



III.3.2 Opérations sur les listes en langage C

- Création et parcours

Création : Pour créer une nouvelle liste, il suffit de déclarer une variable de type liste et d'initialiser sont champ début à NULL.

Une fois qu'on a créé une liste, on veut la remplir. Pour cela, on a besoin de pouvoir créer de nouveaux éléments. Deux possibilités existent :

- Soit on déclare une variable de type element ;
- Soit on effectue une allocation dynamique.

En fin, Il faut donc utiliser la seconde, basée sur malloc. La fonction suivante prend une valeur en paramètre et renvoie un pointeur sur un nouvel element, ou bien NULL si l'allocation dynamique échoue.

```
Element * cree_element(int valeur)
{  element *e;
if((e = (element *) malloc(sizeof(element))) != NULL)
{  e->valeur = valeur;
e->suivant = NULL;
```

```
}
return e;
}
```

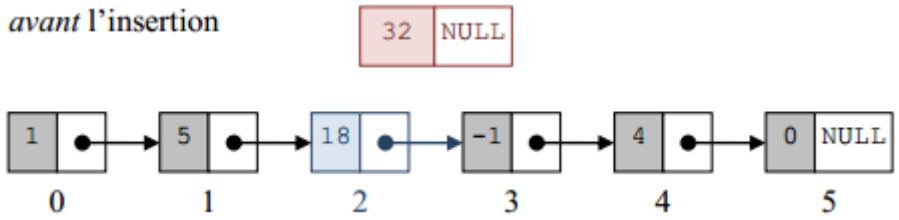
Parcours :

Pour cela, l'approche standard consiste à utiliser pointeur temporaire, que nous noterons temp, et qui ne sera utilisé que pour cette tâche de parcours. Il est initialisé au début de la liste, et est modifié par une boucle (voire une fonction récursive).

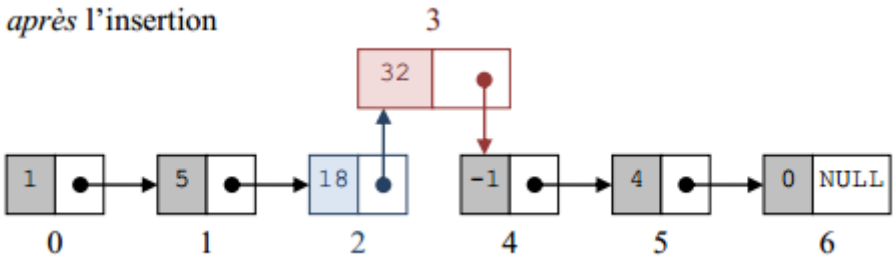
```
void affiche_liste(liste l)
{  element *temp = l.debut;
printf("{");
while(temp != NULL)
{  printf(" %d", temp->valeur);
temp = temp->suivant;
}
printf(" }\n");
}
```

- Insertion d'un élément

Exemple : on veut insérer un nouvel élément de valeur 32 à la position 3 dans la liste de l'exemple précédent :



On doit modifier le pointeur suivant de l'élément précédent (position 2), et celui du nouvel élément.




```
int insere_element(liste *l, element *e, int p)
{  element *temp = l->debut, *avant, *apres;
int erreur = 0;
//  si on insère en position 0, on doit modifier l
if(p==0)
{  e->suivant = temp;
l->debut = e;
}
// sinon, on doit modifier le champ 'suivant' de l'élément précédent
else
{  // on se place au bon endroit
temp = accede_element(*l,p-1);
```

```
// on insère l'élément
if(temp==NULL)
erreur = -1;
else
{  avant = temp;
apres = temp->suivant;
e->suivant = apres;
avant->suivant = e;
}
}
return erreur;
}
```

- **Accès à un élément**

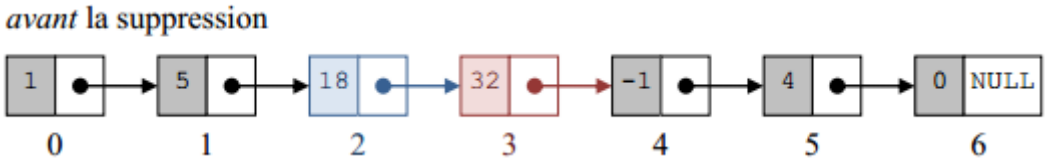
Pour accéder à l'élément situé en position p de la liste, il faut parcourir les $p - 1$ éléments situés avant lui.

```
element* accede_element(liste l, int p)
{  element *resultat = l.debut;
int i;
i=0;
while(i<p && resultat!=NULL)
{  resultat = resultat->suivant;
i++;
}
return resultat;
}
```

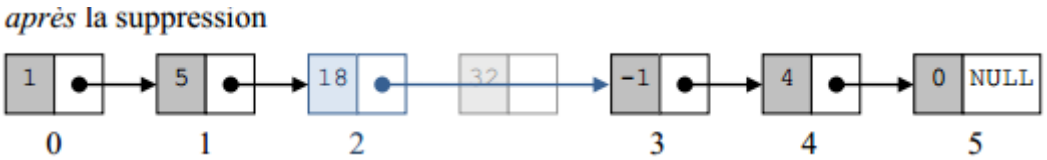
- **Suppression d'un élément**

La suppression d'un élément dans une liste simplement chaînée se fait de façon à peu près symétrique à l'insertion. Il est nécessaire de modifier le pointeur suivant de l'élément qui précède l'élément à supprimer.

Exemple : on veut supprimer l'élément de valeur 32 situé à la position 3 après l'exemple précédent :



On doit modifier le pointeur suivant de l'élément précédent (position 2), de manière à le faire pointer vers l'élément suivant (position 4) de l'élément à supprimer (position 3).



Soit int supprime_element (liste *l, int position) la fonction qui supprime l'élément situé à la position p dans la liste l. La fonction renvoie −1 si elle n'a pas pu supprimer l'élément à la position demandée ou 0 si la suppression s'est bien passée.

```
int  supprime_element (liste *l, int p)
{  element *temp = l->debut, *e, *avant, *apres;
int erreur = 0;
//  si on supprime en position 0, on doit modifier l
if(p==0)
{  e = temp;
if(e == NULL)
erreur = -1;
else
l->debut = e->suivant;
}
// sinon, on doit modifier le champ 'suivant' de l'élément précédent
else
{  // on se place au bon endroit
temp = accede_element(*l,p-1);
// on supprime l'élément de la liste
if(temp==NULL)
erreur = -1;
else
{  avant = temp;
e = temp->suivant;
if(e == NULL)
erreur = -1;
else
{  apres = e->suivant;
avant->suivant = apres;
}
}
}
// on désalloue l'élément
if(erreur != -1)
free(e);
// on termine la fonction
return erreur;
}
```

III.3.1.3 Les piles et les files

Les piles et les files

Ce sont des structures de données ordonnées, mais qui ne permettent l'accès qu'à une seule donnée. Les piles (LIFO : Last In First Out) correspondent à une pile d'assiettes : on prend toujours l'élément supérieur, le dernier empilé.

III.3.1.3.1 **Les files** (FIFO : First In First Out) correspondent aux files d'attente : on prend toujours le premier élément, donc le plus ancien. Elles sont très souvent utiles, elles servent à mémoriser des évènements en attente de traitement.

Opérations primitives sur les files

1. Ajouter () ou Enqueue()

2. Enlever () ou Dequeue()
3. Vide () ou Empty()
4. Remplissage ()

Les applications qui utilisent les files

Intérêts

- Mémoriser temporairement les transactions informatiques qui attendent un traitement
- Serveurs d'impressions, qui doivent traiter les requêtes dans l'ordre dans lesquels elles arrivent, et les insèrent dans une file d'attente.
- Certains moteurs multitâches, dans un système d'exploitation, qui doivent accorder du temps machine à chaque tâche, sans privilégier aucune.

Examinons une façon d'implémenter une fille d'attente, ici rangée dans la classe Queue.
On commence à définir :

```
class Queue{
int fin;
int[] t;
static Queue creer(int taille){
Queue q = new Queue();
q.t = new int[taille];
q.fin = 0;
return q;
}
}
```

La variable fin sert ici à repérer l'endroit du tableau ***t*** où on stockera le prochain client. Il s'ensuit que la fonction qui teste si une queue est vide est simplement :

```
static boolean estVide(Queue q){
return q.fin == 0;
}
```

L'ajout d'un nouveau client se fait simplement : si le tableau n'est pas rempli, on le met dans la case indiquée par fin, puis on incrémente celui-ci :

```
static boolean ajouter(Queue q, int i){
if(q.fin >= q.t.length)
return false;
q.t[q.fin] = i;
q.fin += 1;
return true;
}
```

Quand on veut servir un nouveau client, on teste si la file est vide, et si ce n'est pas le cas, on sort le client suivant de la file, puis on décale tous les clients dans la file.

```
static void servirClient(Queue q){
if(!estVide(q)){
System.out.println ("Je sers le client "+q.t[0]);
for(int i = 1; i < q.fin; i++)
q.t[i-1] = q.t[i];
q.fin -= 1;
```

}}

Un programme d'essai pourra être :

```
class Poste{
static final int TMAX = 100;
public static void main(String[ ] args){
Queue q = Queue.creer(TMAX);
Queue.ajouter(q, 1);
Queue.ajouter(q, 2);
Queue.servirClient(q);
Queue.servirClient(q);
Queue.servirClient(q);
}}
```

III.3.1.3.2 **Les piles** (LIFO : Last In First Out), pour accéder au premier élément, il faut commencer par dépiler le dernier jusqu'au premier.

Opérations primitives sur les piles

1. Empiler () ou Push () i.e Ajouter

L'empilement se fait simplement en rajoutant un élément au début de la liste.

```
int empile (pile *p, int v)
{  int erreur;
element *e;
e = cree_element(v);
if(e == NULL)
erreur = -1;
else
erreur = insere_element(p, e, 0);
return erreur;
}
```

Ou

```
int empile (pile *p, int v)
{  int erreur = 0;
if(est_pile_pleine(*p))
erreur = -1;
else
{  p->contenu[p->taille] = v;
p->taille ++ ;
}
return erreur;
}
```

2. Dépiler () ou Pop () i.e retirer

Le dépilement est réalisé en supprimant le sommet, c'est-à-dire le premier élément de la liste
Le dépilement consiste à :

- Supprimer le dernier élément de notre représentation de la pile ;

- Mettre à jour la taille de la pile.

```
int depile (pile *p)
{  int erreur = 0;
if(est_pile_vide(*p))
erreur = -1;
else
p->taille - - ;
return erreur;
}
```

Ou

```
int depile(pile *p)
{  int erreur = supprime_element(p, 0);
return erreur;
}
```

3. Vide () ou Empty() i.e vide

```
int est_pile_vide(pile p)
{  return (p.debut == NULL);
}
```

4. création

La création de pile revient tout simplement à créer une liste :

```
pile cree_pile()
{  liste l;
l.debut = NULL;
return l;
}
```

5. Accès au sommet

Pour accéder au sommet, il suffit de renvoyer le dernier élément de notre représentation de la pile.

```
int sommet(pile p, int *v)
{  int erreur = 0;
if(est_pile_vide(p))
erreur = -1;
else
*v = p.contenu[p.taille-1];
return erreur;
}
```

6. Remplissage ()

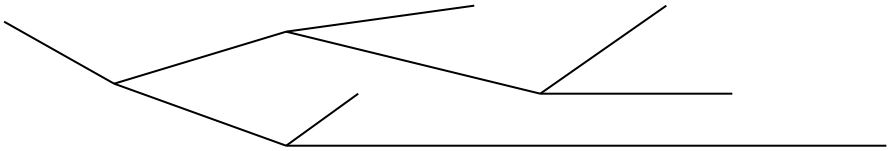
Applications des piles

- La fonction « undo » pour annuler la frappe dans le clavier
- Evaluation des expressions mathématiques (notion de post fixée, notation polonaise)
- Navigateurs Web, ici les piles sont utilisées pour mémoriser les pages Web visités, l'adresse de chaque est empilée.

- Dans les systèmes d'exploitation, pour gérer l'espace mémoire associé aux différents appels de fonction faits au sein d'un programme.
- Dans les compilateurs ou analyseurs syntaxiques, entre autres pour analyser des expressions bien parenthésées.
- En intelligence artificielle, dans les algorithmes d'exploration d'arbres, pour effectuer par exemple un parcours en profondeur.

III.3.1.3.3 **Les arbres**

Les listes sont des structures dynamiques unidimensionnelles. Les arbres sont leur généralisation multidimensionnelle. Une liste est un arbre dont chaque élément a un et un seul fils. Chaque composante d'un arbre contient une valeur et des liens vers ses fils.



III.3.1.3.4 Rechercher dans du texte

Rechercher une phrase dans un texte est une tâche que l'on demande à n'importe quel programme de traitement de texte, à un navigateur, un moteur de recherche, etc. C'est également une part importante du travail accompli régulièrement en bioinformatique.

Vues les quantités de données gigantesques que l'on doit parcourir, il est crucial de faire cela le plus rapidement possible. Le but de cette section est de présenter quelques algorithmes qui accomplissent ce travail.

III.3.1.3.4.1 la recherche Naïve

C'est l'idée la plus naturelle : on essaie toutes les positions possibles du motif en dessous du texte. Comment tester qu'il existe une occurrence en position ***i*** ? il suffit d'utiliser un indice j qui va servir à comparer M[j] à T[i+j] de proche en proche :

```
static boolean occurrence (char[ ] T, char[ ] M, int i){
for(int j = 0; j < M.length; j++)
if(T[i+j] != M[j]) return false;
return true;}
```

Nous utilisons cette primitive dans la fonction suivante, qui teste toutes les occurrences possibles :

```
static void naif (char[ ] T, char[ ] M) {
System.out.print("Occurrences en position :");
for(int i = 0; i < T.length-M.length; i++)
if(occurrence(T, M, i))
System.out.print(" "+ i +",");
System.out.println(" "); }
```

III.3.1.3.4.2 Algorithme linéaire de Karp-Rabin

Supposons que S soit une fonction (non nécessairement injective) qui donne une valeur numérique à une chaîne de caractères quelconque, que nous appellerons signature :

Le principe de l'algorithme de Karp-Rabin utilise cette idée de la façon suivante :on remplace le test d'occurrence T [i…i + m 1] = M [0…m 1] par S(T [i…i + m 1]) = S(M [0…m 1]).

Voici la fonction qui implante cette idée. Nous préciserons la fonction de signature S plus loin (codée ici sous la forme d'une fonction signature) :

```
static void KR(char[] T, char[] M){

int n, m;
long hT, hM;
n = T.length;
m = M.length;

System.out.print ("Occurrences en position :");

hM = signature (M, m, 0);
for(int i = 0; i < n-m; i++){

hT = signature(T, m, i);
if(hT == hM){

if (occurrence(T, M, i))
System.out.print(" "+i+",");
else
System.out.print(" ["+i+"],");
}
}
System.out.println("");
}
```

La fonction de signature est critique. Il est difficile de fabriquer une fonction qui soit à la fois injective et rapide à calculer : La première fonction à laquelle on peut penser est celle qui se contente de faire la somme des caractères représentés par des entiers :

```
static long signature(char[ ] X, int m, int i){
long s = 0;
for(int j = i; j < i+m; j++)
s += (long) X[j];
return s;
}
```

Avec cette fonction, e programme affichera :

Occurrences en position: 10, 13, [18].

III.3.1.3.5 **la table de Hachage**

L'intérêt de ces structures de données, les problèmes qu'elles soulèvent et les moyens disponibles pour résoudre ces problèmes. Disons tout de suite que là où il Ya une masse considérable de données pour lesquelles les structures de données avancées conventionnelles ne sont pas bien adaptées à cause du cout temporel et / ou spatial, les tableaux associatifs comme les tables de hachage peuvent constituer une solution économique.

III.3.1.3.4.1 principe des tables de hachage

De quoi s'agit-il ?

Imaginons la taille de la base de données est très grande, si ***n*** est trop grand cette structure (tableau) n'est pas une bonne solution en terme de temps de l'exécution et de l'espace mémoire.

Une bonne solution ???

Une bonne compromis entre l'espace mémoire et le temps de traitement.

Ce qu'on veut.

Construire une structure de données qui permette l'accès immédiat aux clés d'une base de données à partir d'une modification de données de leurs empreintes (code, index, adresse,…), avec un tableau, il faut que la taille de la structure m << n (la taille des objets).

III.3.1.3.4.2 nécessaires d'une table de hachage

Le problème que l'on rencontre avec l'adressage direct est l'univers U des clés doit être très petit, afin de pouvoir être mis en bijection avec l'intervalle [0, m-1] par h, tout en rendant possible la réservation d'un tableau de taille *m*. On peut néanmoins s'affranchir de cette condition en n'imposant plus que *h* soit une bijection mais seulement une surjection. On dit alors que *h* est une ***fonction de hachage***, et la table une ***table de hachage***. Le problème qui peut alors apparaître est le cas de deux clés c_1 et c_2 telles que $h(c_1)=h(c_2)$, ce qui s'appelle ***une collision***. Il va sans dire que l'on peut même avoir une collision entre plus de deux, mais pour pallier à ces problèmes il existe des algorithmes ou des méthodes efficientes lors de l'ajout, recherche, suppressions,….. D'une clé ***x*** dans la table de hachage, que nous verrons tout on long de ce cours.

Le choix d'une fonction d'adressage (fonction de hachage)

$$H : \begin{cases} U \rightarrow \{0,1,2,3,\ldots\ldots\ldots,m-1\} \\ x \rightarrow h(x) \end{cases}$$

Nota : |U|= n >> m

III.3.1.3.4.3 Comment insérer une clé x dans la table de hachage ?

Il nous faut d'abord choisir une fonction de hachage. Pour l'instant ne cherchons pas à savoir si nous avons une bonne fonction ou une mauvaise. On verra ce problème plus loin.

Voyons comment la fonction de hachage hache les clés pour leur fournir un rang dans la table de hachage.

Clé(x)……HACHAGE……..H(x)=rang de x

III.3.1.3.4.3.1 Insertion

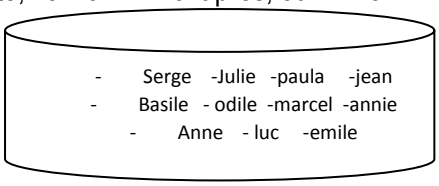
$$H : \begin{cases} U \rightarrow \{0,1,2,3,\ldots\ldots\ldots,12\} \\ x \rightarrow h(x) \end{cases}$$

III.3.1.3.4.3.2 Quel choix pour H ???

Pour calculer H(x)= [x' + n(x)] MOD m

Où : x' : la valeur numérique d'un mot x de la liste (les 26 lettres de l'alphabet français) ; n(x) : le nombre des caractères dans le mot x et m : la taille du table de hachage.

Exemple : Calcule de l'empreinte, voir la BDD ci-après, où m=13



X'=19+5+18+7+5= 54
n(serge)= 5

H (basile)=2 H(Emile)=3 H (serge)=7 H(jean)=8 H(odile)=11
H (luc)=0 H (paula)=4 H(marcel)=6 H (annie)=9 H(julie)=10 H(anne)=12

II.3.1.3.4.3.2 Taux de remplissage

$\propto$=n /m ; où n : les cellules occupées et m : la taille de la table de hachage.

$\propto$= 11/13=85%

NB : on apprendra qu'il n'existe pas de fonction de hachage universelle.

Dans la partie précédente, nous avons utilisé une fonction de hachage pour remplir une table de hachage. Cette fonction avait un bon taux de remplissage et nous n'avions pas eu de collisions. De plus les clés étaient bien réparties dans la table.

Dans cette partie, nous voulons le faire voir des méthodes disponibles pour construire une fonction de hachage et leurs limites. On terminera en tirant des généralités sur les qualités d'une bonne fonction H(x).

III.3.1.3.4.3.3 Méthodes de hachage

1. Méthode par extraction
2. Méthode par compression
3. Méthode par division (modulo)
4. Méthode par multiplication

Qu'est-ce qu'une bonne fonction de hachage ?

III.3.1.3.4.3.3.1 Méthode par extraction

Cette méthode consiste à convertir une clé x en binaire (base : 0/1) sur 5 bits par caractère, puis extraire des bits à certains positions 1, 2, 7, 8 mais en comptant de la droite pour en former une clé x la quelle sera converti en un entier.

Exemple :

Clés	Binaire	Binaire extraire	décimal	Code H(x)
Et	00101 10100	1000	8	H (et)
Ou	01111 10101	1101	13	H (ou)
Ni	01110 01001	1101	13	H (ni)

Il	01001 01100	0000	0	H (il)

D'où il Ya un cas de collision, vu que deux mots différents ont une même adresse, nous allons voir plus tard comment résoudre ces problèmes.

III.3.1.3.4.3.3.2 Avantage de la méthode

Calcul facile à mettre en œuvre

III.3.1.3.4.3.3.3 inconvénients

Adaptée seulement à des cas particuliers quand on connait la valeur des clés à priori et qu'on sait que certains bits ne sont pas significatifs et ne donne pas de bons résultats en général, car ne prend pas la totalité de la valeur de la clé.

III.3.1.3.4.3.3.2 Méthode par compression

Principe : identifier la clé x en sous chaines de même longueur puis à additionner ces sous chaines ; pour éviter les restes, nous allons utiliser le ou-exclusif (xor).

Exemple : hacher la clé et :

Et= 00101 10100

(00101 xor 10100)=$(10001)_{10}$ = 17

H(et)=17 ; H(ou)=26 et H(ni)=7

Remarque : l'opération ou exclusif est commutatif.

H (et) = H(te), d'où il y aura collision.

Avec cette méthode les anagrammes et en fait toutes les permutations d'un mot entrent toujours en collision. Mais il y a une manière d'éviter ces collisions.

Solution : on peut améliorer la fonction de hachage par compression en procèdent à un décalage circulaire d'un bit.

Exemple : comment hacher CAR et ARC sans qu'ils entrent en collision ?

Prenons les clés x_1=CAR et X_2= ARC

x_1= CAR= 00011 00001 10010

Décalage	00011	00001	10010
D'un bit	**10001**	10000	01001
Deux bits		**01000**	10100
Trois bits			**01010**

H(CAR)=10001 xor 01000 xor 01010=$(10011)_{10}$=19

X_2= ARC=00001 10010 00011

Decalage	00001	10010	00011
D'un bit	**10000**	01001	10001
Deux bits		**10100**	11000
Trois bits			**01100**

H(ARC)=10000 xor 10100 xor 01100=$(01000)_{10}$= 8

Nous avons résolu les collisions en effectuant les décalages circulaires.

III.3.1.3.4.3.3.3 méthode par division modulo

Méthode très simple et très rapide !!!

$$H : \begin{cases} U \rightarrow \{0,1,2,3, \ldots \ldots \ldots, m-1\} \\ x \rightarrow h(x) \end{cases}$$

H(x)= x' mod m

Où : x' : la valeur numérique d'un mot x de la liste (les 26 lettres de l'alphabet français) et m : la taille du table de hachage.

Exemple : soit la clé, prenons par exemple m=13 et x=naima

X'=14+1+9+13+1=38 ; pour trouver l'empreinte de cette clé H(naima)= 38, mod 13=12

Cette méthode pose des problèmes liés à la valeur de m ; en général pour éviter les collisions. Pour obtenir de bons résultats.

- ***m*** est un nombre premier
- Il faut éviter des entiers tels que $m=2^k$

Exemple : m=12, marche par ce qu'il ne peut pas s'écrire sous forme 2^k.

Avantage : facile et rapide à calculer

Inconvénient : dépend trop de m. Par exemple si m est pair alors tous les éléments(les clés) vont aller dans les indices pairs de la table hachage T[ ] ; si m est impair alors tous les éléments vont aller dans les indices impaires de la table hachage t[ ].

Si (m) à des petits diviseurs, le hachage ne produit pas une bonne uniformité dans la table. Il faudrait que la taille m de T soit un nombre premier mais même dans le cas il peut y avoir des phénomènes d'accumulation.

III.3.1.3.4.3.3.4 méthode par multiplication

III.3.1.3.4.3.3.4.1 Principe : (l'algo à revoir)

1. On se donne $\theta \in]0,1[$
2. On se donne ***m*** la taille de la table hachage
3. H(x) = ε[[num(x)* θ) mod 1]*m] ; ε :la partie entière

On montre que la méthode par multiplication donne de bon résultat

$$\theta=\begin{cases} \frac{\sqrt{5}-1}{2} \\ 1-\frac{\sqrt{5}-1}{2} \end{cases} \qquad \theta=\begin{cases} 0{,}6180 \\ 0{,}38 \end{cases}$$

NB : 1-φ ; les nombres d'or dans la suite de fibonnaci.

Exemple : on demande de hacher la clé ET prendre m=30 et thêta=0,6125423371

num(et)=5+20=25

H(et)= ε[[(25*0,6125423371) mod 1]* 30]

= ε[(15,31355843 mod 1)*30]

= ε[9,40]= 9

H(te)= ε[[(25*0,6125423371) mod 1]* 30] // attention à revoir,il suffit des gérere les deux methodes.

III.3.1.3.4.3.3.4.2 critères d'une bonne fonction de hachage

Il n'existe pas en pratique une meilleure méthode universelle.

III.3.1.3.4.3.3.4.3 Qualités d'une bonne fonction de hachage

1. Rapide à calculer
2. Repartir uniformément les clés de U dans la table de Hachage
3. Bon taux de remplissage

Dans la partie suivante, nous étudierons des méthodes permettant de gérer les collisions. Ces méthodes sont de deux types

1. Les méthodes indirectes (chainage)

   Avec cette méthode, les clés se trouve dans une même clé, la table de hachage serait équivalent à une seule liste chainée dont les éléments seront respectivement les clés(x).

   1.1 chainage séparé
   1.2 chainage coalescent

2. Les méthodes directes (calculs)

   Ici au lieu que les clés en collisions sont dans une même liste, cette méthode vont calculées un nouvel emplacement pour chaque collision ; ces calculs peut être linéaire ou double.

   2.1 Hachage linéaire
   2.2 Hachage double

Nota : Démonstration à titre d'exercices

III.3.1.3.6 Complexité algorithmique

Comme on l'a vu lors de l'introduction du cours, il est souvent possible d'implémenter plusieurs solutions à un problème donné. La question se pose alors de savoir quel algorithme est le plus efficace.

Et pour effectuer cette comparaison, il est nécessaire de déterminer sur quels critères elle doit être effectuée.

Le but de cette section est d'introduire le champ de la théorie de la complexité, dont l'un des buts est de répondre à ce type de question.

III.3.1.3.6.1 Motivation et principe

Théorie de la complexité : théorie développée dans le but de comparer efficacement différents algorithmes entre eux, Afin de savoir lequel était le meilleur pour un problème donné.

Jusqu'aux années 70, on caractérisait les algorithmes de manière empirique, en considérant :

- Le temps d'exécution ;
- L'espace mémoire utilisé ;
- Les conditions d'exécution :
  - Taille des données
  - Type d'ordinateur
  - Système d'exploitation
  - Langage de programmation
  - Etc.

Exemple :

Pour le problème consistant à calculer 12^{34}, sur un P4 à 3 GHz, avec un OS Windows XP et une implémentation en langage C, supposons que l'algorithme A met 1,2 secondes et utilise 13 Mo de mémoire, alors que l'algorithme B met 1,3 secondes et utilise 0,8 Mo. Alors on peut dire que A est plus rapide que B, mais que B est moins gourmand en mémoire.

Le problème de cette démarche est qu'elle ne permet pas de comparer les algorithmes de manière objective, car ils ne sont pas les seuls facteurs intervenant dans la performance de résolution du problème : d'autres facteurs extérieurs interviennent également (matériel, logiciel...).

La solution retenue est de se détacher de ses facteurs extérieurs, et donc de l'implémentation de l'algorithme. Pour cela, nous allons utiliser une approche plus théorique, reposant sur deux notions : les opérations élémentaires et les positions mémoire.

Opération élémentaire : opération atomique, correspondant à une instruction assembleur. Rappelons qu'une instruction assembleur est atomique, dans le sens où elle correspond à un code directement interprétable par le processeur.

Position mémoire : unité de mémoire élémentaire, correspondant généralement à un octet. La notion de position mémoire est une abstraction de l'occupation qu'un algorithme a de la mémoire. Elle va nous permettre d'évaluer cette consommation de façon indépendante de certaines conditions d'exécution mentionnées précédemment, en considérant le nombre de positions mémoires qu'il utilise.

NB : Ces deux notions nous permettent d'exprimer la complexité d'un algorithme en fonction de la taille des données qu'il traite.

Taille des données d'un problème : entier(s) représentant la grandeur des paramètres reçus par l'algorithme devant résoudre le problème.

Complexité d'un algorithme : nombre d'opérations élémentaires ou de positions mémoire dont l'algorithme a besoin pour résoudre un problème d'une certaine taille.

III.3.1.3.6.2 Complexités spatiale et temporelle

Complexité temporelle : nombre total d'opérations élémentaires pour exécuter l'algorithme.

La complexité temporelle correspond donc au décompte de toutes les opérations élémentaires que l'algorithme a besoin d'effectuer pour résoudre le problème.

Complexité spatiale : nombre maximal de positions mémoire utilisées au cours de l'exécution.

Remarque : il existe un lien entre les complexités spatiale et temporelle d'un algorithme. D'abord, sur une machine monoprocesseur (i.e. ne possédant qu'un seul processeur, et donc capable d'exécuter une seule instruction à la fois), la complexité spatiale ne peut pas dépasser la complexité temporelle, puisqu'il faut effectuer au moins une opération pour initialiser une position mémoire.

III.3.1.3.6.3 Cas favorable, moyen et défavorable

Complexité dans le meilleur des cas : complexité de l'algorithme quand les données traitées aboutissent à la complexité minimale. On parle aussi de cas favorable.

Complexité dans le pire des cas : complexité de l'algorithme quand les données traitées aboutissent à la complexité maximale. On parle aussi de cas défavorable.

Complexité moyenne : complexité de l'algorithme moyennée sur l'ensemble de toutes les données possibles.

EXERCICES :

1) Ecrire un algorithme permettant de renvoyer la plus petite et la plus grande valeur en précisant sa position dans un vecteur linéaire.

2) Écrivez une fonction entier longueur_liste(liste l) qui renvoie le nombre d'éléments présents dans la liste l passée en paramètre. longueur_liste(liste l) : entier.

3) Donner un algorithme d'une fonction qui retourne le produit C de deux matrices carrées d'ordre n passé en argument et ensuite retourne la plus petite valeur sur la nouvelle matrice C.

4) Ecrire un sous-programme appel-sommation () et appel tri, qui crée et initialise un tableau linéaire en demandant à l'utilisateur la taille du tableau à créer et les différentes valeurs à y ranger. Cet algorithme fera ensuite appel à la fonction sommation et affichera son résultat.

5) Soit un tableau S(25, 4). On suppose que Base(S)=200 et qu'il existe w=4 mots par case mémoire. On suppose également que le langage de programmation range le tableau bidimensionnel dans l'ordre par ligne prioritaire. Déterminer l'adresse de S(12, 3).

6) On donne un tableau à trois dimensions :
A(1 :4 ;2 :6 ;3 :8) déterminer l'adresse de l'élément A(2,3,4) étant que la base(A)=200 et le nombre de mot w par case mémoire vaut 4. Si on suppose que :

a) A est rangé selon un ordre ligne prioritaire
b) A est rangé selon un ordre colonne prioritaire.

7) Soit un tableau « Etudiants » de type enregistrement, dont la structure est : Numéro Matricule, Nom de l'étudiant, Post Nom, Sexe, Age, Section, Promotion. Proposez un algorithme qui permet de créer ce tableau, puis de le compléter et enfin de visualiser tous ses éléments.
8) On se propose de trier une liste de noms de manière à les ranger par ordre alphabétique. On utilise la méthode de tri ; écrire un algorithme réalisant ce tri.

9) Ecrire un algorithme qui permet d'insérer à bonne position un nombre quelconque dans une liste chaînée monodirectionnelle d'ancrage s triée par ordre croissant.

10) Soit une liste chaîne unidirectionnelle d'ancrage S. Ecrire une procédure qui supprime tous les éléments de rang pair.

11) Soit une liste linéaire suivante : 60, 42, 11, 51, 33, 56, 41, 67, 15;

a) Appliquer l'algorithme de tri par insertion pour arranger cette liste par ordre croissant
b) Appliquer l'algorithme de recherche binaire à la liste trier IT= 60.

12) On demande de Hacher les mots ci- après : In,xor, data et info dans une base de donnée.

1. En utilisant la méthode de hachage par multiplication et par compression, dans les cas des collisions veuillez améliorer.
2. Evaluer le taux de remplissage.

13) Qu'est-ce qu'une bonne fonction de hachage.

14) Ecrire un algorithme effectuant une recherche dichotomique dans un tableau linéaire trié avant recherche, en utilisant le tri par sélection ; ensuite illustrez par un exemple.

15) Soit une liste linéaire suivante : 7, 5, 13, 3, 6, 9

a) Appliquer l'algorithme de tri par segmentation pour arranger cette liste par ordre croissant.
b) Illustrez par un graphe la trace cet algorithme.

16) Evaluer la complexité des différents algorithmes de tri, des recherches et de fusion dans un tableau linéaire.

17) Ecrire un algorithme qui permette de dire si un mot donnée est un polindrome, c'est-à-dire identique à son miroir.
Exemple : esse,maxam,kayak,…

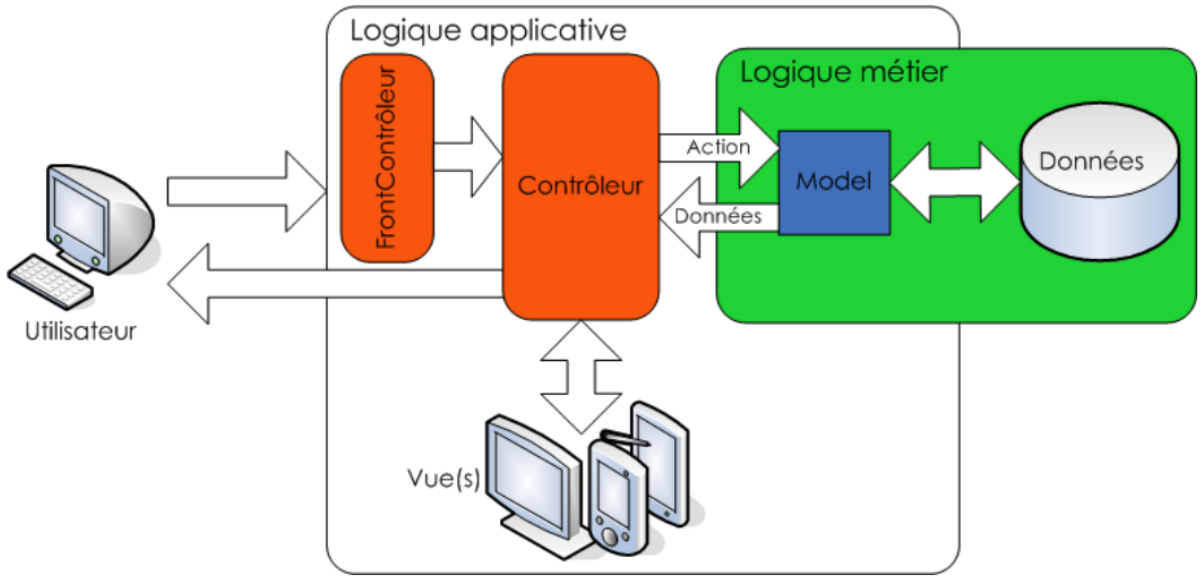
ANNEXE SUR LA PROGRAMMATION EN COUCHES

Nous allons utilisé le modèle vue contrôleur (MVC) comme notre design pattern.

Le MVC est très utilisé dans les applications informatiques, Il est décomposé en 3 parties :

- Le contrôleur : Il récupère les données du formulaire pour les traiter
- Le modèle : qui gère la structuration et l'accès aux données
- La vue : qui gère la partie 'présentation' (visible)

Son architecture se présente comme suit :



- La vue : Elle gère l'aspect du formulaire, son design
- Le contrôleur : Il récupère les données du formulaire pour les traiter
- Le modèle : Il fait l'interface entre la base de données d'une part, le contrôleur et la vue d'autre part

TRAVAUX PRATIQUE :

Objectif de travaux pratique dans le cours d'algorithmique et méthodes de programmation, sont destinées aux exposés sur l'usage de l'étude approfondie du cours, nous avons jugé utilisé de donner aux étudiants de G2 Informatique, une connaissance théorique et pratique sur l'utilisation et le comportement en arrière-plan des langages de programmation.

Série1 : Mettre en place un algorithme et un programme en Basic, vb.net, C#, java ou sur un langage de votre choix.

1) L'algorithme d'ecriture binaire en décimal
2) L'algorithme d'ecriture décimal en binaire
3) Tri par insertion
4) Tri par bulles
5) Tri par sélection
6) Tri fusion
7) Tri par segmentation
8) La dérivée d'un polynôme
9) La recherche séquentielle
10) La recherche binaire dans un vecteur
11) La recherche binaire dans une matrice
12) Implémentez un programme permettant de trouver une valeur choisie aléatoirement par le programme. Le joueur disposera au maximum de 3 tentatives pour trouver cette valeur et le programme lui indiquera à chaque essai si sa valeur est trop grande ou trop petite.
13) Implémentez un programme d'une fonction récursive qui demande une chaîne à l'utilisateur et qui affiche à l'écran son inverse, le point d'arrêt : si la chaine est vide, plus d'appel récursif, on retourne une chaine vide.

Série 2 :

1) Faites une étude approfondie sur le modèle MVC.
2) Faites une étude approfondie sur la programmation en couches.
    - Couche IHM
    - Couche DAO
    - Couche Entité

Références

1. Kris Jamsa, C/C++/C# La Bible du programmeur, Ed. Reynald Goulet inc., 2004.
2. Marion Marchand IFT-17588 (A-05), Chapitre 2, Fondements de l'analyse des algorithmes
3. Clavel G, et Biondi J., Introduction à la programmation, Tome 1 : Algorithme et langages, masson, 1987
4. Eugene MBUYI MUKENDI, cours d'algorithmique et structure de données, UNIKIN, 2015.
5. Damien Berthet, Vincent Labatut, Algorithmique programmation en langage C, 2015.

Exemple pratique d'un programme en vb.net : de la conversion d'un nombre entier en Binaire.

```
Module Module1

    Sub Main()

        Console.Out.WriteLine(".........Convesion Décimal-Binaire.....  ")
        Console.Out.WriteLine("")
        Console.Out.WriteLine("introduire une valeur:")
        Dim d As Integer = CType(Console.In.ReadLine(), Integer)

        Dim p As String = ""

        If d Mod 2 = 0 Then
            'traitrement nombre pairs
            While (d > 0)
                p &= (d Mod 2)
                d = d \ 2
            End While

            Console.Write("Binaire est:" & david(p))

        Else
            'traitement nombre impairs
            Dim dav As String = ""
            While (d > 0)
                'Console.Write(d Mod 2)
                dav &= (d Mod 2)
                d = d \ 2
            End While
            Console.Write("Binaire est:" & (dav))
        End If
        Console.Out.WriteLine("")
        Console.Out.WriteLine("----------------------------------------")
        Console.Out.WriteLine("         Merci Ir. NSIAMUNU DAVID ")
        Console.Out.WriteLine("........................................")
        Console.Out.WriteLine("...Appui sur une touche, pour quitter....")
        Console.ReadKey()
    End Sub
    'traitement inverse de la chaine
    Function david(ByVal st As String) As String
        If st = "" Then
            david = ""
        Else
            david = st.Substring(st.Length() - 1, 1) + david(st.Substring(0,
st.Length() - 1))
        End If

    End Function
End Module
```