Algorithmique et Structures de Données

POLYCOPIE DE COURS

Cours et Travaux Dirigés
Support destiné aux étudiants de niveau
Première et deuxième année Licence

Dr. Mourad AMAD
Enseignant au Département d'Informatique
Faculté des Sciences Exactes
Université Abderrahmane Mira de Bejaia

Année 2016

Avant Propos

Ce polycopié est rédigé à l'intention des étudiants de première et de deuxième année du premier cycle universitaire (*licence*). Il constitue un manuel de cours et d'exercices sur une partie du domaine de programmation. Les lecteurs ne nécessitent aucun pré requis sur les l'algorithmique.

Ce polycopié est structuré en huit chapitres comme suit :
Dans le premier chapitre, des notions de base sur la structure globale d'un algorithme sont données, ainsi que les différentes parties qui le composent suivie par les instructions de base les plus élémentaires.
Le deuxième chapitre décrit en détails les différentes structures de contrôles (*boucles*) qui peuvent être utilisées dans un algorithme (*ex. Pour, tant que, ..*).

Le chapitre trois aborde l'utilisation des tableaux dans la programmation. Le quatrième chapitre est consacré aux sous programmes (*fonctions et procédures*). Dans le cinquième chapitre, l'utilisation des enregistrements et des fichiers dans le cadre de l'algorithmique est expliquée.
Le sixième chapitre traite la récursivité afin de faciliter l'écriture des algorithmes qui peuvent être récursifs. Dans le septième chapitre, nous avons illustré comment calculer la complexité algorithmique de n'importe quel algorithme.
Enfin, le chapitre huit est consacré à la programmation dynamique. La notion de pointeur est illustrée. Des modèles de programmation pour les listes linéaires chainée, les files, les piles ainsi que les arbres sont donnés.

Une liste de références bibliographiques est donnée à la fin de ce manuscrit.

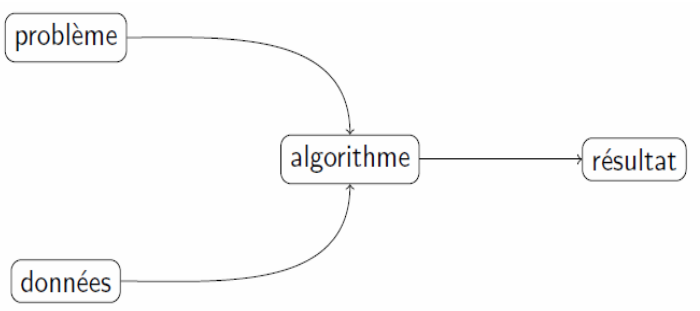
Sommaire

Chapitre 1 :

Généralités et Notions de Base

1. Introduction

L'algorithmique est l'étude des algorithmes. Un algorithme est une méthode permettant de résoudre un problème donne en un temps fini ;

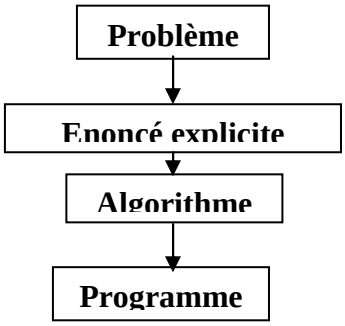


Un algorithme est une suite de raisonnements ou d'opérations qui fournit la solution d'un problème. Le programme ne sera que la traduction de l'algorithme dans un langage de programmation, c'est-à-dire, un langage plus simple que le français dans sa syntaxe, sans ambiguïtés, que la machine peut utiliser et transformer pour exécuter les actions qu'il peut décrire. Pascal, C, Java et Visual Basic sont des noms de langages de programmation

LAROUSSE : « un ensemble de règles opératoires dont l'enchaînement permet de résoudre un problème au moyen d'un nombre fini d'opérations. »

Réalisation d'un programme

La résolution d'un problème donné passe par une succession d'étapes à savoir :



La réalisation d'un programme passe par l'analyse descendante du problème : il faut réussir à trouver les actions élémentaires qui, en partant d'un environnement initial, nous conduisent à l'état final. Une fois ces actions déterminées, il suffit de les traduire dans le langage de programmation choisi.

Durant l'écriture d'un programme, on peut être confronté à 2 types d'erreur :

1. **Les erreurs syntaxiques** : elles se remarquent à la compilation et sont le résultat d'une mauvaise écriture dans le langage de programmation.
2. **Les erreurs sémantiques** : elles se remarquent à l'exécution et sont le résultat d'une mauvaise analyse. Ces erreurs sont beaucoup plus graves car elles peuvent se déclencher en cours d'exploitation du programme.

2. Base d'un langage algorithmique

Le langage algorithmique est un langage générique permettant de traiter tous type de problème par la concaténation des instructions.

2.1 Structure de Base

La structure générale d'un algorithme (*Programme*) est la suivante :

1. **Algorithme** Nom-d'Algorithme ;
2. déclaration des variables et des constantes
3. Déclaration des fonctions
4. **Début**
5. …..
6. Liste des instructions
7. ……..
8. **Fin.**

2.2 Variables et constantes

Une variable est un espace mémoire nommé de taille fixe, prenant au cours de déroulement de l'algorithme un nombre indéfini de valeurs différentes. Ce changement de valeur se fait par l'opération d'**affectation** notée **(←).** La variable diffère de la notion de constante qui, comme son nom l'indique, ne prend qu'une valeur unique au cours de l'exécution de l'algorithme.

La partie déclaration permet de spécifier quelles seront les variables utilisées au cours de l'algorithme ainsi que le type de valeur quelles doivent respectivement prendre.
Parmi les types des variables les plus utilisés, on trouve :

2.2.1 Entier : (1,2, 3,….)
Le type entier caractérise l'ensemble des nombres entiers. Les opérations arithmétiques possibles sur ce type sont : L'addition '+', '-', '*', '/'. Appliquées sur des opérandes de type entier, elles fournissent un résultat entier sauf l'opération '/' qui fournit un résultat réel. Tandis que les opérations de relation notées par '<', '>', '>=', '<=', '<>' fournissent un résultat logique.

Exemples :
Const x = 1 ;
Var A : entier ;
Var A, b : entier ;
Var coefficient_module : entier ;

2.2.2 Réel : (flottant : 1.0, 1.35, 1E+12, …)
Représente l'ensemble des nombres réels. Les opérations '+','-', '*', '/' appliqués sur des opérandes de type réel fournissent un résultat de type réel.

Exemples
Var A : réel ;
Var A, b : réel ;
Var moyen_générale : réel ;

2.2.3 Caractère : (A, a, Z, …)
Ce type englobe tout les caractères nécessaires à l'écriture de texte.

Exemples :
Const x = 1.1 ;
Var A : **Caractère** ;
Var A, b : **Caractère** ;
Var xxxxxxxxxxxx : **Caractère**;

2.2.4 Booléen : (Vrai, Faux)
Ce type de variable ne peut prendre que les valeurs vrai ou faux. Les operateurs logiques ET, OU, NON, appliqués à des opérandes de type booléen fournissent un résultat booléen.

Exemples :
Const x = 'a' ;
Var A : **Booléen** ;
Var A, b : **Booléen**;
Var Admis : **Booléen**;

2.3 Identificateurs et mots clés
→ Un identificateur est le nom d'une variable, d'une constante ou le nom d'un algorithme (programme), c'est un seul mot composé de chiffres et des caractères à l'exception de quelques uns.

→ Un mot clé est un mot réservé, il a une utilisation spéciale dans un programme comme par exemple : program, begin, end, if, else, case, repeat, until, for, do, while, then, var, …

→ Règles d'écriture des identificateurs
1. Un identificateur ne peut être un mot clé
2. Un identificateur doit commencer par un caractère alphanumérique
3. Un identificateur ne doit pas contenir des caractères spéciaux comme : ?, !, *, …

Exemple
Quels sont les identificateurs valides et ceux qui ne sont pas valides ?
M, ax, 8b, s_m, farid, exo1, exo ?, 34, then, nom programme, ….

2.4 Instructions
Une instruction est une action élémentaire commandant à l'ordinateur un calcul, une instruction de base peut être :

A : Une affectation ou une opération arithmétique
L'affectation est l'action élémentaire principale puisque c'est par son intermédiaire que l'on peut modifier la valeur d'une variable, elle a pour syntaxe : **Variable← valeur** ou **variable ← expression ;**

Exemple

Algorithme **exemple-Affectation ;**
Var A, B : **Entier ;**
Debut
……..
A← 10 ;
B← A+15 ;
Fin.

B : Cas d'utilisation d'une affectation

→Pour modifier la valeur d'une variable

…
A←10 ;
B←5 ;
A←A+B ;
…

→Pour affichage sur écran

L'affichage est l'action élémentaire permettant à un algorithme de fournir des résultats à l'utilisateur, il se fait par l'intermédiaire de la commande (*Fonction*) **Ecrire**.

…
A←10 ;
B ←5 ;
C←A+B ;
Ecrire (« j'ai calculé la somme de A+B ») ;
…

→Pour une lecture au clavier

La lecture au clavier est une action élémentaire permettant de spécifier par une intervention humaine la valeur d'une variable. La saisie se fait par l'intermédiaire de la commande (*Fonction*) **Lire.**

…
A←10 ;
Lire (B) ;
C←A+B ;
…

Exercice 1 : (*Savoir déchiffrer une séquence d'instructions***)**

Dite que fait l'ensemble des instructions suivantes

…
A←1 ;
B←10 ;
C←A*2+B-3 ;
Ecrire (C) ;
….

2.5 Commentaires

Un commentaire est un texte facultatif (*des phrases*) situé entre les symboles et { et } qui n'a aucun effet sur le fonctionnement du l'algorithme. Elle sert à expliquer le pourquoi de certaines instructions utilisés dans un programme.

Exemple
Algorithme exo ;
Var A : entier ;
Debut
/* ce programme calcul la somme de deux nombre*/
Lire (A) ;
Lire (B) ;
C :=A+B ;
Ecrire (C) ;
Fin.

2.6 Expressions

Une expression est une combinaison de plusieurs opérandes (*éventuellement des fonctions*) et opérandes. Ils existent plusieurs types d'expressions :

- ➔ Les expressions arithmétiques
  Elles font intervenir les operateurs arithmétiques (+, -, /, *) ainsi que les deux operateurs DIV et MOD.
  Exemple : a + b – c/d est une expression arithmétique
- ➔ Les expressions booléennes (logiques)
  Elles font intervenir les operateurs logiques (and, or, not, …) ainsi que les operateurs de relation (>, <, <=, <>, …).
  Exemple : (a > b et b <= c) et une expression logique

Priorité des operateurs :
Une expression est évaluée suivant la priorité des operateurs, on commençant par le plus prioritaire. Dans le cas ou deux operateurs ont la même priorité, on commence par la plus à gauche.

• / div		Non (not)
+ - mod	Priorité	et (and)
		Ou (or)

<, > , <>, >=, <=, ont la même priorité.

Les parenthèses ouvrantes et fermantes sont considérées des operateurs les plus prioritaires.

2.7 Operateurs MOD et DIV

L'opérateur MOD fournit la partie entière de la division entre deux opérandes
Exemple : a ← b div 5 ;
L'operateur MOD fournit le reste de la division entière entre deux opérandes
Exemple : a ← b mod 5 ;

3. Conclusion

Ce chapitre permet de se familiariser avec le langage algorithmique afin d'écrire des petits algorithmes pour résoudre des petits problèmes. Cependant, les instructions vues dans ce chapitre sont omniprésents dans tous les futurs algorithmes.

Série D'exercice N1

Exercice 1
Ecrire L'algorithme (*l'ensemble des actions en langage naturel*) qui permet d'écrire la méthode de résolution d'une équation de deuxième degré dans l'ensemble R.

Exercice 2
Considérons le problème suivant : Une personne veut retirer d'une citerne pleine d'eau, une quantité de capacité égale 4 litres sachant qu'elle possède seulement un seau de 3 litres et un autre de 5 litres. Comment peut-elle faire ?

Exercice 3
Quel est le type de chaque variable :
A=1, B=vrai, test= 12.23, specialite='m',

Exercice 4
Soient A, B deux variables de types entiers, C, D deux variables de type réel, E, F deux variables de type booléen.
Quel est le type des variables suivants : A1, B1, C1, A2, B2, C2, D2, A3, B3, C3, D3

A1← A+B ; B1← A*B; C1←A/B;
A2← C+D; B2← C*D; C2← C/D; D2← A*C;
A3← E ou F; B3← E et F; C3← (A>B) ; D3← Faux ;

Exercice 5
Quel sont les identificateurs valides et ceux qui ne sont pas valides :
A, cA, 12, 1A, A1, A12m, bougie, test?, SI,

Exercice 6
Faite le déroulement de program suivant en donnant la valeur finale de chaque variable

```
Program Exo5 ;
Var  E : real ;
     A, B, D : integer ;
Begin
A :=2 ;
B :=3 ;
D := A*B+5 ;
E := D/2 ;
End.
```

Exercice 7
Quelle sont les erreurs dans le programme suivant :

```
Program Sinon;
Var  E : reél ;
     A, B, C : entiers ;
     1m := réel ;
Début
A :=2 ;
B :=3 ;
C :=(A+B)/2 ;
```

1m :=C*B ;
Fin.

Exercice 8
Ecrire un programme qui permet d'introduire un nombre et d'afficher son double ainsi que sa moitié.

Exercice 9
Soient a =2, b=3, c=9, d=11, q=7 ;
Evaluer l'expression E dans les trois cas suivants:

1. E← a + b * c - d ;
2. E← b * d / q - a ;
3. E← - a + b / 2 ;

Exercice 10
Ecrire l'instruction E équivalente à chacune des expressions suivantes :
a) sans faire les simplifications mathématiques et
b) après avoir fait les simplifications nécessaires

1. $\dfrac{a+b/c}{\dfrac{c}{b}+\dfrac{a-4}{a}}$
2. $a+8-\dfrac{c+a*b}{a+\dfrac{a}{3}}$
3. $\dfrac{a*b+c}{a-b}+\dfrac{a}{c}-4$

Chapitre 2 :

Les Structures de Contrôle

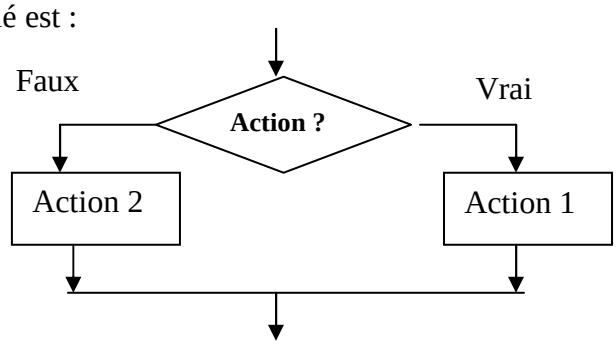
Introduction
On appel structure de contrôle toute action qui permet d'effectuer un groupe d'actions sous condition(s), et permet donc d'orienter le déroulement d'un programme suivant la réalisation de ces conditions.
Il existe plusieurs formes de structure de contrôle :

1. L'action conditionnelle ou alternative
Sa forme est :

Si <Condition> **alors** <Action 1>
Sinon <Action 2 >
Fsi

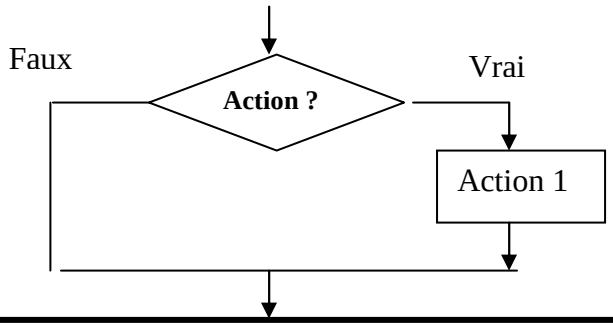
Cette action est s'exécute de la manière suivante :
→ Si la condition définie dans <Condition> est vérifiée il faut exécuter l'action (*ou le groupe d'action*) définie dans <Action 1>, puis les actions qui suivent le fsi
→ Si cette condition n'est pas vérifiée il faut exécuter l'action (*ou le groupe d'actions*) définie dans <Action 2>, puis les actions qui suivent le fsi.

L'organigramme associé est :



Il existe une seconde forme de cette action conditionnelle :
Si <Condition> **alors** <Action 1> **Fsi**

Et dans ce cas :
→ Si la condition <Condition> est vérifiée, l'action (*ou le groupe d'actions*) <Action 1> est exécutée puis les actions qui suivent le fsi.
→ Par contre si la condition n'est pas vérifiée, seules les actions qui suivent le fsi seront exécutées.
L'organigramme associé est :

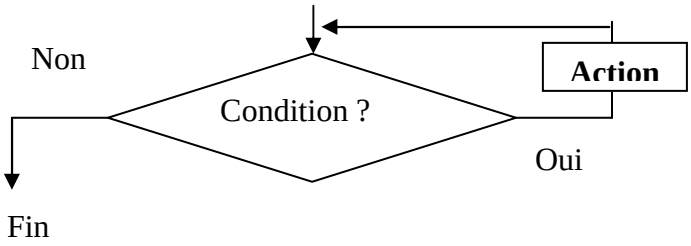


2. L'action répétitive 'Tant que'
Sa forme est :

Tant que <Condition> **faire**
<Action>
Fait ;

Cette action spécifie qu'une action (*ou groupe d'actions*) doit être répétée tant que la condition reste vérifiée.

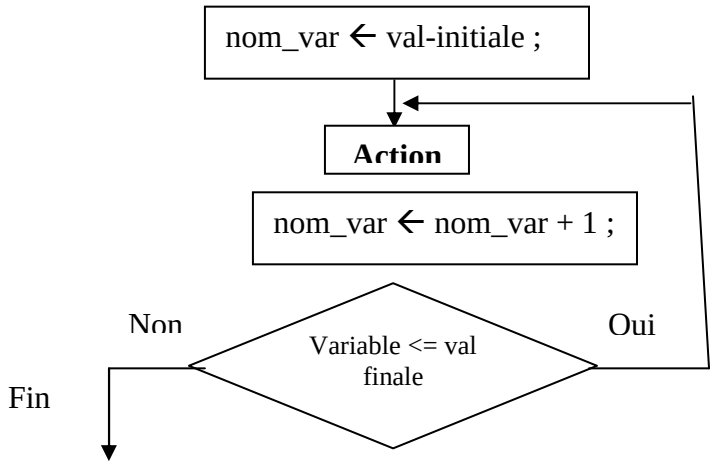
Remarque : L'action <Action> étant répétée jusqu'à ce que la condition devienne fausse, il faut donc que la partie action soit capable de modifier les paramètres intervenant dans la condition **condition** afin de sortir de tant que.



3. L'action répétitive 'pour'
Sa forme est :

Pour <nom_var> **allant de** <val_initiale> **à** <val_finale> **faire**
<Action>
Fait ;

Cette action permet de répéter une action (*ou groupe d'actions*) un nombre de fois déterminé par les valeurs initiale et finale de paramètre <nom_var>. La partie action est exécutée la première fois en affectant au paramètre <nom_var> que l'on appelle aussi compteur, la valeur <val_initiale>. Apres chaque exécution on rajoute au compteur la valeur 1. L'action est ré exécuté jusqu'à ce que la valeur du compteur dépasse la valeur spécifiée par <val_finale>.

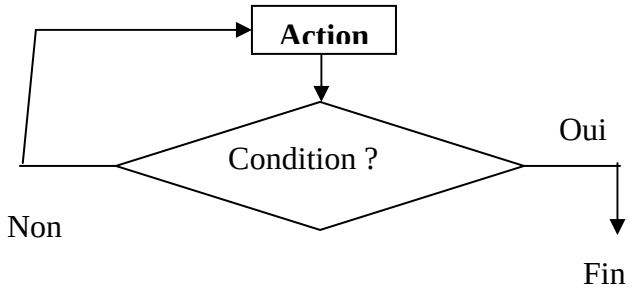


4. L'action répétitive 'répéter--------jusqu'à'

Cette action permet de répéter une action (*groupe d'actions*) jusqu'à ce que la condition soit vérifiée. Sa forme est la suivante :

Repeter
<Actions>
Jusqu'à (condition)

Remarque : L'action est exécutée au moins une fois à l'inverse de l'action répétitive «tant que » ou l'action ne peut être exécutée



5. L'action conditionnelle à choix multiple

Sa forme est la suivante :
Case variable **of**
1 : Action 1 ;
2 : Action 2 ;
Default: Action n;
Fin;

Cas_i : peut être une constante ou plusieurs constantes séparées par des virgules ou bien un intervalle

Exemple :
Cas ch **of**
'a'..'z' : ecrire ('lettre') ;
'0'..'9' : ecrire ('chiffres') ;
'+', '-', '*', '/' : ecrire ('operation') ;
sinon
ecrire ('caractere speciale');
fin,

Conclusion

Les structures de controles sont tres importantes pour controler le deroulement des algorithmes, chacune est adequate pour une situation donnée afin d'optimiser l'execution des algorithmes.

Série D'exercices N2

Exercice 1
Ecrire le programme qui permet d'interchanger les valeurs de deux variables entières. **a)** avec utilisation d'une variable intermédiaire, **b)** sans utilisation d'une variable intermédiaire.

Exercice 2
Ecrire le programme qui permet de faire l'ordonnancement de deux nombres entiers A, B, // (A<B), ensuite le programme d'ordonnancement de trois nombres entiers //(A<B<C).

Exercice 3
Ecrire le programme qui cherche le maximum (max) et le minimum (min) entre deux nombres donnés a et b ensuite entre trois nombres donnés a, b et c.

Exercice 4
Ecrire le programme qui demande à l'utilisateur un nombre *n* compris entre 1 et 3 jusqu'à ce que la réponse convienne.

Exercice 5
Ecrire le programme qui demande un nombre de départ *n*, et qui ensuite écrit la table de multiplication de ce nombre *n*;

Exercice 6
Ecrire le programme qui demande un nombre de départ *n*, et qui calcule la somme des entiers jusqu'à ce nombre. Par exemple, si l'on entre n=5, le programme doit calculer :

EX : 1 + 2 + 3 + 4 + 5 = 15

Exercice 7
Ecrire le programme qui permet le calcul de la somme de N nombres donnés (n1, n2, …n20).

Exercice 8
Ecrire les programmes qui permettront de calculer les sommes suivantes :

1) $\sum_{i=1}^{n} \sum_{j=1}^{i} (i + j)$ **2)** $\sum_{i=1}^{N} X^i$

Exercice 9
Ecrire le programme qui calcul le PGCD (*le plus grand diviseur commun*) et le PPMC (*le plus petit multiple commun*) de deux nombres A, B donnés

Exercice 10
Ecrire les programmes qui permettront de calculer : le factoriel de N ! , ainsi que le nombre de combinaisons de *p* parmi *n* C_N^P

Exercice 11

Ecrire le programme qui demande successivement 20 nombres à l'utilisateur, et qui lui dit ensuite quel était le plus grand parmi ces 20 nombres :

Exercice 12

Ecrire le programme qui permet de calculer la somme des N premiers termes d'une suite géométriques définie par : **U0** (*premier terme*), **R** (*Raison*).

Exercice 13

Ecrire le programme qui permet d'introduire un nombre entier inferieur ou égal à 10 et le réaffiche en lettres

Ex :

1	Un
2	Deux
3	Trois

Exercice 14

Dans une entreprise, le calcul des jours de congés payés s'effectue de la manière suivante : si une personne est entrée dans l'entreprise depuis moins d'un an, elle a droit à deux jours de congés par mois de présence, sinon à 28 jours au moins. Si c'est un cadre et s'il est âgé d'au moins 35 ans et si son ancienneté est supérieure à 3 ans, il lui est accordé 2 jours supplémentaires. S'il âgé d'au moins 45 ans et si son ancienneté est supérieure à 5 ans, il lui est accordé 4 jours supplémentaires, en plus des 2 accordés pour plus de 35 ans. Écrire le programme qui calcule le nombre de jours de congés à partir de l'âge, l'ancienneté et l'appartenance au collège cadre d'un employé.

Exercice 15

Ecrire le programme qui permet d'introduire un nombre entier *n* et de calculer et d'afficher son inverse.

Ex : 1234 ----------→ 4321

Exercice 16

Ecrire le programme qui permet de calculer le N-ième terme U_N de la suite de FIBONACCI qui est donnée par la relation de récurrence: $U_1=1$ $U_2=1$ $U_N=U_{N-1} + U_{N-2}$ (*pour N>2*)

Exercice 17

Un nombre entier est dit **parfai**t s'il est égal à la somme de ses diviseurs à part lui-même. **Exemple** : 6 = 1+2+3, donc 6 est un nombre parfait. Ecrire le programme qui permet de vérifier si un nombre donné *n* est parfait ou non.

Exercice 18

On souhaite à partir d'un capital de départ et d'un d'intérêt mensuel, calculer le montant obtenu après une période d'un certain nombre de mois. Par exemple si le capital déposé est de 150, et le taux d'intérêt mensuel de 2%, alors au bout d'un mois, les intérêts acquis sont de 3 (= 150*0.02) et le nouveau capital est de 153 ; au bout de 3 mois, les intérêts acquis sont de 9,1812 et le nouveau montant est de 159,1812. La période est donnée par l'utilisateur sous la forme de deux dates, une date de début correspondant au dépôt du capital de départ, et une date de fin de période. Chaque date est donnée sous forme d'un numéro de mois et d'un numéro d'année. Ecrire le programme qui, lorsqu'on lui donne le capital de départ, le taux mensuel, la date de début et la date de fin, calcule le capital obtenu ainsi que les intérêts acquis à la fin de la période donnée.

Chapitre 3:

Les Tableaux

1. Introduction

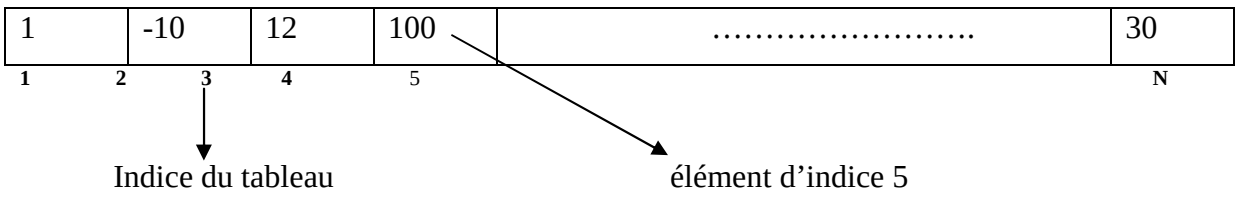
Imaginons que dans un programme, nous avons besoin simultanément de 12 valeurs (*par exemple, des notes pour calculer une moyenne*). Evidemment, la seule solution dont nous disposons à l'heure actuelle consiste à déclarer douze variables, appelées par exemple : Notea, Noteb, Notec, etc. Bien sûr, on peut opter pour une notation un peu simplifiée, par exemple N1, N2, N3, etc. Mais cela ne change pas fondamentalement notre problème, car arrivé au calcul, et après une succession de douze instructions « **Lire** » distinctes, donnera obligatoirement une atrocité du genre :

```
Moy ← (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12 ;
```

C'est pourquoi la programmation nous permet de rassembler toutes ces variables en une seule, au sein de laquelle chaque valeur sera désignée par un numéro. Autrement dit, cela donnerait donc quelque chose du genre « la note numéro 1 », « la note numéro 2 », « la note numéro 8 ». C'est largement plus pratique.

2. Tableau à une dimension (*vecteur*)

Un tableau à une dimension est un ensemble de valeurs portant le même nom de variable, et repérées par un nombre appelé **indice,** et qui sert à repérer chaque valeur du tableau séparément. A chaque fois qu'on doit designer un élément du tableau, on fait figurer le nom du tableau, suivi de l'indice de l'élément entre deux crochets. Le schéma ci-dessous représente un tableau à une dimension.



2.1. Déclaration des tableaux à une dimension

Un tableau à une dimension est déclaré de la manière citée ci-dessous, en précisant le nombre et le type de valeurs qu'il contiendra.

Exemples

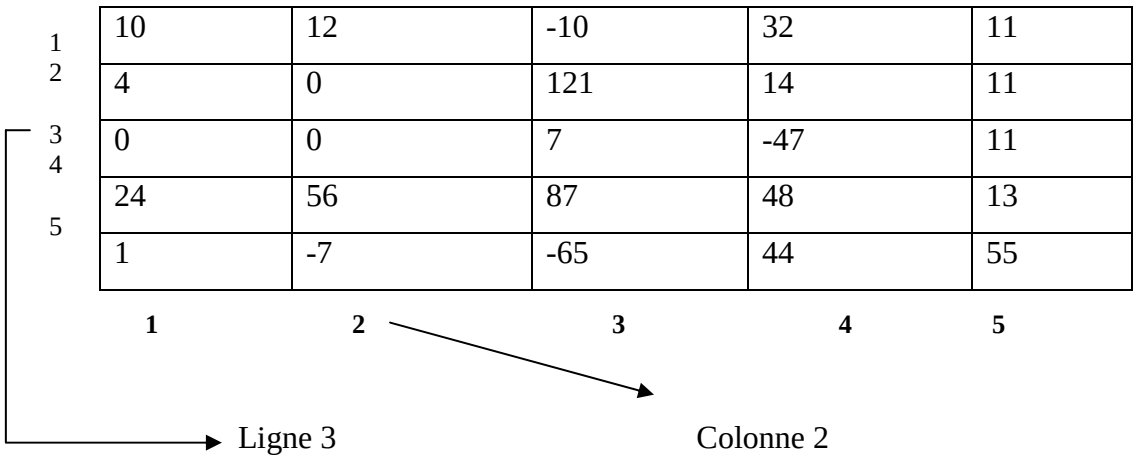
Tab : tableau [1..100] d'entiers ;
Note : tableau [1..10] de réels ;

2.2. Remarques

1. Les cases du tableau (*éléments*) sont numérotées à partir de 1,
2. Lors de déclaration du tableau, on précise la plus grande valeur de l'indice, qui est le nombre d'éléments du tableau

3. Tableau à deux dimensions (*Matrice*)

L'algorithmique (*la programmation*) nous offre la possibilité de déclarer et d'utiliser des tableaux dans lesquels les valeurs ne sont pas repérées par un seul indice comme les tableaux à une seule dimension, mais par deux indices (*coordonnées*), le premier indice sert à représenter les lignes et le second indice pour les colonnes. Le schéma ci-dessous représente un tableau à deux dimensions.



3.1. Déclaration des tableaux à deux dimensions

Un tableau à deux dimensions est déclaré de la manière citée ci-dessous, en précisant le nombre (*lignes et colonnes*) et le type des valeurs qu'il contiendra.

Exemples

Achat : tableau [1..100, 1..100] d'entiers ;
Vente : tableau [1..10, 1..47] de réels ;

Le tableau de la figure précédente peut être déclaré de la manière suivante :

Taexemple : tableau [1..5,1..5] d'entiers ;

4. Tableaux à N dimensions

Un tableau à N dimensions est déclaré de la manière citée ci-dessous, en précisant le nombre et le type de valeurs qu'il contiendra.

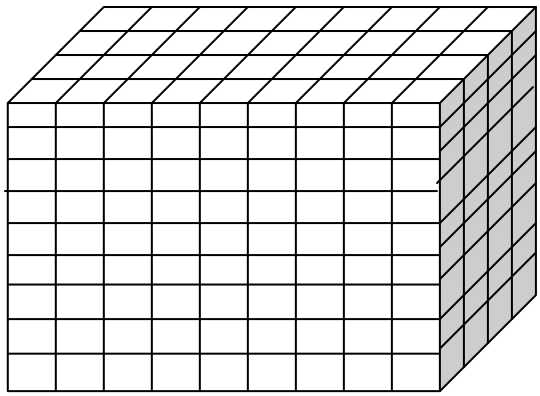
Exemples

Achat : tableau [1..100, 1..100, 1…300] d'entiers ;

Vente : tableau [1..10, 1..47, 1..12, 1..7] de réels ;

Le premier est un tableau à trois dimensions et le deuxième est un tableau à quatre dimensions.

La référence d'un élément dans un tableau à N dimensions, se fait de la même manière que celle pour un tableau à une ou à deux dimensions, en précisant l'indice de chaque dimension.



Exemple d'un tableau à trois dimensions

5. Conclusion

Les tableaux sont des structures de programmation statiques, la taille de tableau doit être précisée à l'avance. Un espace mémoire peut être réservé sans être utilisé si un tableau n'est pas exploité complètement. Un tableau, une fois déclaré, sa taille ne peut pas être modifiée.

Série D'exercices N3

Exercice 1
Ecrire un programme qui permet de **remplir** un tableau de N éléments, et **d'afficher** son contenu

Exercice 2
Ecrire un programme qui permet de **rechercher** un élément donné X dans un tableau

Exercice 3
Ecrire un programme qui permet de rechercher le **minimum** et le **maximum** dans un tableau de N éléments

Exercice 4
Ecrire un programme qui permet de calculer la **moyenne** des éléments d'un tableau.

Exercice 5 (*Tri par Bulle***)**
Ecrire un programme qui permet de **trier** les éléments d'un tableau par la méthode de bulle. La méthode de bulle consiste à balayer tout le tableau, en comparant les éléments adjacents et les échangeant s'ils ne sont pas dans le bon ordre. Un seul passage ne déplacera un élément donné que d'une position, mais en répétant le processus jusqu'à ce plus aucun échange ne soit nécessaire, le tableau sera trié.

Exercice 6 (*Recherche séquentiel d'un élément dans un tableau***)**
Il suffit de lire le tableau progressivement du début vers la fin. Si le tableau n'est pas trié, arriver en fin du tableau signifie que l'élément n'existe pas. Dans un tableau trié le premier élément trouvé supérieur à l'élément recherché permet d'arrêter la recherche

Exercice 7 (*Recherche dichotomique d'un élément dans un tableau***)**
Ecrire un programme qui permet de **rechercher** un élément donné par la méthode dichotomique dans un tableau à une dimension trié.

Exercice 8
Ecrire un programme qui permet de **remplir** et d'**afficher** un tableau à deux dimensions

Exercice 9

Ecrire un programme qui permet de chercher le **minimum** dans un tableau à deux dimensions

Exercice 10

Ecrire un programme qui permet de calculer la moyenne des éléments de l'**anti diagonale** d'un tableau à deux dimensions

Exercice 11
Soient **A** un tableau qui contient les notes de tous les modules d'un étudiant X, **B** : Le vecteur des coefficients qui correspond à ces notes, Ecrire un programme qui permet de calculer la **moyenne** de l'étudiant X.

Exercice 12
Ecrire un programme qui permet de calculer **le nombre d'occurrences** d'un nombre donné x dans un tableau à deux dimensions

Exercice 13
Ecrire un programme qui permet de calculer la **transposé** d'une matrice (*tableau à deux dimension*) : $\mathbf{B = A^t}$

Exercice 14
Ecrire un programme qui permet de calculer le **nombre des éléments** (*valeurs*) **pairs** et le **nombre des éléments impairs** dans un tableau à deux dimensions.

Chapitre 4 :

Les Fonctions et les Procédures

1. Introduction

Les fonctions et les procédures sont appelés aussi les sous-programmes, elles fournissent aux programmeurs un moyen simple d'abstraction en lui permettant de nommer une séquence d'instructions et de l'appeler autant de fois qu'il sera nécessaire au cours d'un même programme. En outre, les langages de programmation tels que Pascal, permettent de passer des données à la procédure ou la fonction. Les valeurs de ces données pouvant variés d'un appel à l'autre, ce mécanisme est appelé passage de paramètres. Un sous-programme (*fonction ou procédure*) à la même structure qu'un programme, les seules différances importantes sont les notions de paramètres et de variables locales.

2. Fonctions

Une fonction est un ensemble d'instructions qui forment un sous programme, les fonctions en algorithmique (*programmation*) ressemblent à celles de mathématique, Chaque fois que l'ont appelle elle renvoie au programme appelant une valeur qui est le résultat du traitement effectués par des instructions de la fonction. Une fonction peut renvoyer n'importe quel type de base.

2.1 Déclaration des fonctions

La structure générale d'une fonction est comme suit :

Fonction NomFonction (Var1 : **type1**, Var2 : **type2**, …VarN : **typeN**) : **Type** ; {l'entete de la fonction}
Var variable1, …VariableN : **type** ; {déclaration des variables propres à la fonction}
Begin
Instructions ; {corps de la fonction}
End ;
Où : Nomfonction est le nom de la fonction (Identificateur)
Type1, Type2, …,TypeN sont les type des paramètres de la fonction
Type est le type de résultat retourné.
Instructions est le calcul effectué par la fonction.

Remarque : À la fin d'une fonction, il faut affecter le résultat du traitement au nom de la fonction.

2.2 Utilisation des fonctions

Une fonction, une fois déclarée, pourra être appelée depuis le programme principal, ou une autre fonction.

Exemple : Calcul de Cnp
Algorithme CalculCNP ;
Var cnp**,** n, p **: entiers**

Fonction fact (n : entier) : entier
Var t, i : entiers ;
Debut
t←1 ;
Pour i allant de 1 **à** n **faire**
Debut
t←t*i ;
Fin ;
Fact←p ; {le résultat doit être affecté au nom de la fonction}
Fin ;

Debut
Ecrire (‘donner la valeur de n et p’) ;
Lire (n,p) ;
Cnp← fact(n)/(fact(n-p)*fact(p)) ;
Ecrire (‘cnp=’,cnp);
Fin.

2.3 Variables locales et variables globales

Les variables qui sont déclarées dans la partie déclaration de la fonction s’appellent variables locales. Elles sont inconnues par le programme principal ou la fonction à été déclarée. Les variables qui sont déclarées dans le programme principal ne sont pas connues par les fonctions déclarées dans le même programme, ils sont appelées variables globales.

2.4 Paramètres formels et paramètres réels (effectifs)

Les paramètres décrites dans l’entête de la fonction lors de déclaration sont appelées paramètres formels, ils servent à représenter la structure et le fonctionnement de la fonction. Par contre ceux qui les remplacent dans les appels des fonctions sont appelées paramètres effectifs ou réels.

2.5 Appel d’une fonction

Pour appeler une fonction, on fait apparaître le nom de la fonction à droite d'une affectation de variable souvent appelée variable de retour. La syntaxe est :
Variable ← Nom_fonction (paramètres) ;

3 Procédures

Le deuxième type de sous-programme est appelé procédure. La procédure ressemble parfaitement à la fonction sauf que cette dernière peut renvoyer zéro ou plusieurs résultats.

3.1 Déclaration d’une procédure

Une procédure est déclarée dans la partie déclaration d’un programme de la manière suivante ;
Procedure Nom_procedure (Var1 : **type11**, Var2 : **type12**, …VarN : **type1N**, Res1 : **Type21**, Res2 : **type22**, …,Res2N : **type2N**)
Begin
Liste des instructions; {le corps de la procédure}
End;

Où : Nom_procedure est le nom de la procédure (*Identificateur*)
Var1, var2, … : sont les paramètres d'entré,
Res1, Res2, … : sont les paramètres de sortie (*les résultats fournis par la procédure*)
Typeij : est le type de paramètre associé ;

Remarques

1. les dernières instructions dans une procédure sont généralement réservées pour affecter les résultats de traitement effectué par la procédure aux paramètres de sortie de cette dernière.
2. Les notions de variables globales, locales, les paramètres formels et effectifs pour les procédures ont les mêmes significations que pour les fonctions.
3. L'appel d'une procédure se fait en écrivons son nom et en remplaçons les paramètres formels par les paramètres effectifs.

2 Fonctions prédéfinies

Ils existent certaines fonctions et procédures prédéfinies, et intégrées avec les langages de programmation (Pascal). Citons à titre d'exemple : **Read** (*pour la lecture des donnés*), **write** (*pour l'affichage des résultas*), **Textcolor** (numéro) (*pour modifier la couleur de texte*), **gotoxy (x,y)** (*pour positionner le curseur au point de coordonnées x, y au moment de l'exécution*), **clrscrn** (*pour effacer l'écran*), **sound** (numéro) (*pour déclencher un bip sonore*), …

3 Conclusion

Les sous programmes (*fonctions et procédures*) sont des portions des programmes qui se répètent souvent soit dans un même programme, soit dans plusieurs programmes, il est donc préférable de les écrire une fois pour toute, et de les utiliser au besoin. Ils aussi à simplifier le développement des programmes volumineux.

Série D'exercices N4

Exercice 1

Ecrire l'algorithme (*programme*) qui permet de calculer C_N^P

Exercice 2

Ecrire l'algorithme (*programme*) qui permet de calculer $\sum_{i=1}^{N}(R^i + 1/P^{i+1})$

Exercice 3

Ecrire l'algorithme (*programme*) qui permet de calculer la moyenne et l'écart type des éléments d'un tableau à une dimension donné

Exercice 4

Ecrire l'algorithme (*programme*) qui permet de compter le nombre de chiffres qui composent un nombre entier ainsi que la somme de ces chiffres

Exercice 5

Ecrire un algorithme (*programme*) qui permet de calculer la somme des maximums des lignes d'un tableau à deux dimensions.

NB : Utiliser une fonction qui reçoit le numéro de ligne comme paramètre et retourne le maximum dans cette ligne

Exercice 6

Ecrire l'algorithme (*programme*) qui permet de calculer $\sum_{i=1}^{N} R^i!$

Exercice 7

Ecrire l'algorithme (*programme*) qui permet de trouver à partir de la date actuelle et la date de naissance, l'âge d'une personne et de l'afficher sous forme (*nombre de jours, nombre de mois, nombre d'années*)

Exercice 8

En utilisant des fonctions/procédures, écrire un algorithme (*programme*) qui permet de calculer séparément les sommes suivantes :

$$Som\ \ 2 = \sum_{i=1}^{N} (i! + 3^i)$$

$$Som\ \ 1 = \sum_{i=1}^{N} (i! + \frac{1}{i})$$

Solution de la Série D'exercices N4

Exercice 1

Ecrire l'algorithme (*programme*) qui permet de calculer C_N^P

Solution 1

```
Program exo1,
Var n,p : integer,
      Cnp : real ;
Function fact (x: integer): integer;
Var f, i: integer;
Begin
f:=1;
For i:= 1 to x do
f:=f*i;
Fact:= f;
End;

Begin
Write ('donner n et p');
Read (n,p) ;
Cnp := fact(n)/(fact(n-p)*fact(p));
Write('le cnp=', cnp)
End.
```

Exercice 2

Ecrire l'algorithme (*programme*) qui permet de calculer $\sum_{i=1}^{N}(R^i + 1/P^{i+1})$

Solution 2

```
Program exo2 ;
Var  som :real ;
       i, R, P, N :integer;
Function puiss (x,y : integer) : integer ;
Var j, f: integer;
Begin
For j :=1 to y do
F:=f*x;
Puiss:=f;
End;

Begin
Write ('donner R, P et N');
Read(R, P, N);
Som:=0;
For  i:= 1 to N do
Begin
Som:=som+ puis (R,i)+ 1/puis(P, i+1);
End,
Write ('la somme est:', som)
End.
```

Exercice 3

Ecrire l'algorithme (*programme*) qui permet de calculer la moyenne et l'écart type des éléments d'un tableau à une dimension donné

Solution 3

```
Program exo3;
Var moy, ecrt, :real;
Taille, j: integer;
Tab: array[1..100] of real;
Procedure moy_ecart(taille1, var moy1: real, var ecrt1:real);
Var s1,s2:real;
i:integer
Begin
S1:=0;
For i:=1 to taille1 do
S1:=s1+tab[i];
Moy1:=s1/taille1;

S2:=0;
For i:=1 to taille1 do
S2:=s2+(tab[i]-moy1)* (tab[i]-moy1);
S2:=SQRT(s2);
ecrt1:=s2/SQRT(taille1);
end;

begin
write ('donner la taille du tableau');
read (taille) ;
write ('remplissage du tableau') ;
for j :=1 to taille do
read(tab[j]) ;
moy_ecart(taille, moy, ecrt);
write ('la moyenne des éléments du tableau est', moy) ;
write ('l''ecart type des éléments du tableau est', ecrt) ;
end.
```

Exercice 4

Ecrire l'algorithme (*programme*) qui permet de compter le nombre de chiffres qui composent un nombre entier ainsi que la somme de ces chiffres

Solution 4

```
Program exo4 ;
Var som, nb, n : integer ;
Procedure Nb_Som(n1:integer ; var som1:integer ; var nb1:integer);
Var  n2, som2, quot, rest, cpt: integer;
Begin
n2:=n1;
som2:=0;
Repeat
```

```
        Quot:=n2 div 10;
        Rest:=n2 mod 10;
        Cpt:=cpt+1;
        Som2:=som2+rest;
        n2:=quot;
Until (quot=0);
Som1:=som2;
nb1:=cpt;
end;

begin
write ('donner n')
read (n);
Nb_Som (n,som,nb);
Write ('le nombre de chiffres est', nb);
Write ('la somme de chiffres est', som) ;
End.
```

Exercice 5

Ecrire un algorithme (*programme*) qui permet de calculer la somme des maximums des lignes d'un tableau à deux dimensions.

NB : Utiliser une fonction qui reçoit le numéro de ligne comme paramètre et retourne le maximum dans cette ligne

Solution

```
Program exo5 ;
Var som : real ;
I,j, nbl, nbc : integer;
Tab: array[1..100, 1..100] of real;

Function max (nl, nbc1:integer): real;
Var max1: real;
K:integer;
Begin
Max1:=Tab[nl,1];
For  k:=1 to nbc1 do
Begin
If(tab[nl, k]>max1) then max1:= tab[nl, k];
End;
max:=max1;
End;

Begin
Write ('donner le nbre de ligne et le nbre de colonne');
Read (nbl, nbc) ;
Write('remplissage du tableau') ;
For i := 1 to nbl do
Begin
For j:=1 to nbc do
Begin
```

```
Read (tab[i,j]);
End;
End;
Som:=0;
For i:=1 to nbl do
Som:=som+ max (i, nbc);

Write ('la somme est', som);
End.
```

Exercice 6

Ecrire l'algorithme (*programme*) qui permet de calculer $\sum_{i=1}^{N} R^i!$

Solution 6

```
Program exo6 ;
Var  R, N, i: integer;
Som: real;

Function puiss (x,y : integer) : integer ;
Var j, f: integer;
Begin
For j :=1 to y do
F:=f*x;
Puiss:=f;
End;

Function fact (x: integer): integer;
Var f, i: integer;
Begin
f:=1;
For i:= 1 to x do
f:=f*i;
Fact:= f;
End;

Begin
Som :=0 ;
For i :=1 to N do
Som:=som+ fact(puiss(R,i));
Write ('la somme est ', som);
End.
```

Exercice 7

Ecrire l'algorithme (*programme*) qui permet de trouver à partir de la date actuelle et la date de naissance, l'âge d'une personne et de l'afficher sous forme (*nombre de jours, nombre de mois, nombre d'années*)

Exercice 8

En utilisant des fonctions/procédures, écrire un algorithme (*programme*) qui permet de calculer séparément les sommes suivantes :

$$Som\ \ 2 = \sum_{i=1}^{N} (i! + 3^{i})$$

$$Som\ \ 1 = \sum_{i=1}^{N} (i! + \frac{1}{i})$$

Solution 8

```
Program exo8s2 ;
Var n, i :integer ;
Som2: real;
Function puiss (x,y : integer) : integer ;
Var j, f: integer;
Begin
For j :=1 to y do
F:=f*x;
Puiss:=f;
End;

Function fact (x: integer): integer;
Var f, i: integer;
Begin
f:=1;
For i:= 1 to x do
f:=f*i;
Fact: =f;
End;

Begin
Write ('donner n');
Read (n);
Som2:=0;
For i:=1 to n do
Som2:=som2+fact(i)+puiss(3,i);

Write ('la somme est', som2) ;
End.

Program exo8s1 ;
Var n, i :integer ;
Som1: real;

Function fact (x: integer): integer;
Var f, i: integer;
Begin
```

```
f:=1;
For i:= 1 to x do
f:=f*i;
Fact:= f;
End;

Begin
Write (‘donner n’);
Read(n);
For i:=1 to n do
Som1:=som1+fact(i)+1/i;

Write (‘la somme est’, som1);
End.
```

Chapitre 5 :

Les Enregistrements et les Fichiers »

Introduction

Jusqu'à présent, les informations utilisées dans nos programmes ne pouvaient provenir que de deux sources : soit elles étaient inclues dans l'algorithme lui-même, par le programmeur, soit elles étaient entrées en cours de route par l'utilisateur. Mais évidemment, cela ne suffit pas à combler les besoins réels des informaticiens. Les fichiers servent à stocker des informations de manière permanente, entre deux exécutions d'un programme. Car si les variables classiques, qui sont des adresses de mémoire vive, disparaissent à chaque fin d'exécution, les fichiers, eux sont stockés sur des périphériques à mémoire de masse (*disquette, disque dur, CD Rom…*).

1. Le type chaine de caractère

Formellement une chaîne de caractères est une suite de zéro, un ou plusieurs caractères accolés.

Remarques :

1. La chaîne formée d'aucun caractère s'appelle la **chaîne vide**.
2. Un constant de type caractère peut être vue comme une constante chaîne de caractères de longueur (*taille*) 1.
3. Une chaîne de caractères se déclare au moyen du mot réservé **string,**

Exemples:

- **string** [10] (* chaîne de taille égale à 10 *)
- **string** [1] (* la plus petite chaîne possible *)
- **string** [255] (* la plus grande chaîne possible *)

Les **constantes** d'un type chaîne sont écrites entre apostrophes:

'Abcd' chaine de taille et de longueur courante 4
' ' chaîne vide (taille = longueur courante = 0)
'x' chaine de taille et de longueur courante égale à 1. C'est aussi un constant caractère

2. Enregistrement

On peut regrouper au sein d'une même structure des informations n'ayant pas nécessairement toutes le même type dans une structure appelée enregistrement (**Record**), par exemple, les différentes informations (*nom, prénom, sexe, nombre d'enfant, …*) relatives à un employé d'une entreprise. C'est un type de données défini par l'utilisateur.

Exemple

```
Type personne = record
Nom : string[30] ;
Prenom : string[30] ;
```

Sex : **char** ;
Nb_enfant : **integer** ;
End ;

Ces variables (*enregistrement ou record*) pouvant être manipulé champ par champ de la même façon que les variables de type simple (*entier, caractère,* …). Pour ce faire, un champ d'une variable de type enregistrement est désigné par le nom de la variable, suivi d'un point et du nom du champ concerné.

Exemple
Var employe : type personne ;
Employe. prenom := 'karim' ;

3. Fichier

3.1. Définition

Des fois, il est nécessaire de conserver certaines données après la fin du programme les ayant créées, ceci pour une utilisation future, les fichiers ont été conçus pour ces fins.

Un **fichier** est une structure de données toutes de même type mais dont le nombre n'est pas connu à priori. L'accès à un élément (*à une donnée*) du fichier se fait :

→ Séquentiellement c'est-à-dire en parcourant le fichier élément par élément depuis le début jusqu'à l'élément choisi
→ Directement en donnant la position de l'élément cherché
→ Selon une clé chaque valeur de la clé désignant un élément on obtient ainsi l'élément désiré

Les fichiers sont conservés en **mémoire secondaire** (*disques et bandes magnétiques, disquettes, flash disk...*), les données qui les constituent reste toujours tant que cette mémoire secondaire n'est pas effacée ou endommagée. Chaque fichier est désigné par un nom et possède des attributs tels que date de création, taille, icône...

Il existe deux types de fichiers :

1. **Les fichiers binaires :** Contenant du code binaire représentant chaque élément. Ces fichiers ne doivent être manipulés que par des programmes!
2. **Les fichiers texte :** appelés aussi imprimables, contenant des caractères et susceptibles d'être lus, éditées, imprimés....

3.2 . Operations sur les fichiers

3.2.1 Création

Pour créer un fichier il faut d'abord définit sa structure : son nom logique et physique, …

Exemple
1) Type note = fichier de réel ;
2) Type info_etud = structure
Mat : string[10] ;
Nom : string [15] ;
Prenom : string[15] ;
Note: reel;
Fin;

Type etudiant = fichier d'étudiant;

Le fichier sera reconnu par l'algorithme (*programme*) par un nom dit logique. Le nom logique est attribué au fichier dans la partie déclaration au moyen d'une variable de type fichier.

Exemples
1) **var** fichier1 : etudiant ;
2) **var** fichier2 : fichier info_etud ;
3) **var** fichier3 : fichier de caractere ;

Un fichier n'est reconnu par le SGF (*système de gestion des fichiers*) que par un nom physique, et par conséquent, un nom physique est associé au nom logique du fichier. L'affectation (ou l'association) du nom logique au nom physique est réalisée par la procédure prédéfinie **ASSIGN** dont la syntaxe est :

ASSIGN (nom logique, nom physique) ;
Exemple :
ASSIGN (fichier1, 'c:\etud.txt') ;

Cela veut dire que fichier1 est le fichier etud.txt qui est stocké dans la racine de la partition C :

3.2.2 Ouverture d'un fichier

Pour pouvoir agir sur le contenu d'un fichier existant il faut l'ouvrir, il y a deux modes d'ouvertures des fichiers :

a) Ouverture en mode écriture : cette opération revient à créer un nouveau fichier et à le préparer pour l'écriture, cela est effectué par la procédure prédéfinie **Rewrite**
Exemple : Rewrite (fichier1) ;

Remarque : si cette procédure est appliquée sur un fichier déjà existant, son contenu sera perdu.

b) Ouverture en mode lecture : c'est une opération qui permet d'ouvrir un fichier stocké sur disque pour la lecture et la modification grâce à la procédure **RESET**.
Exemple : Reset (fichier1) ;

3.2.3 Fermeture d'un fichier

Une fois l'utilisation d'un fichier est terminée, il faut le fermer par la procédure **CLOSE**.
Exemple : Close (Fichier1) ;

3.2.4 Lecture et écriture d'un enregistrement dans un fichier

La lecture d'un enregistrement à partir d'un fichier existant s'effectue au moyen de la procédure **READ.**
Read (fichier2, enregistrement) ;
L'écriture d'un enregistrement dans un fichier s'effectue au moyen de la procédure **Write**.
Write (fichier1, enregistrement) ;

3.2.5 Autres opérations

a) Suppression d'un fichier : pour supprimer un fichier c'est la procédure **ERASE**
Erase (fichier1) ;

b) Changement du nom physique : pour changer le nom physique d'un fichier c'est la procédure **RENAME.**
Rename (fichier1, nom physique2) ;

c) Consultation d'un fichier
Les enregistrements (contenu) d'un fichier peuvent être lus de deux manières :

→ **Accès séquentiel :** parcourt du fichier de premier élément au dernier (*fin du fichier*). La fin du fichier est détectée par la fonction booléenne **EOF,** qui retourne la valeur vrai si la fin du fichier, faux sinon.
Exemple : si EOF (fichier1) **alors**

→ **Accès direct :** il est possible d'accéder directement à un élément du fichier en précisant son numéro d'ordre. Ceci est réalisé par la procédure **SEEK**.
Exemple : SEEK (nom logique, position) ;

→ **Accès Indexé :** pour simplifier, il combine la rapidité de l'accès direct et la simplicité de l'accès séquentiel (*en restant toutefois plus compliqué*). Il est particulièrement adapté au traitement des gros fichiers, comme les bases de données importantes.

d) Suppression des enregistrements : Il existe deux techniques pour la suppression des enregistrements à partir d'un fichier : suppression logique et suppression physique.

→ **Suppression logique :** dans la définition de la structure des enregistrements du fichier, on réserve un champ qui indique l'état de l'enregistrement (*supprimé ou non*), ainsi la suppression logique consiste à mettre ce champ à la valeur appropriée.

→ **Suppression physique :** elle consiste à recopier le fichier dans un autre fichier en ne prenons que les enregistrements qui ne sont pas logiquement supprimés. A la fin, on supprime l'ancien fichier tout entier.

Remarque
Apres une série de suppressions logiques. La suppression physique est recommandée.

Conclusion

Les fichiers sont les seules structures de données qui permettent la sauvegarde permanant des données, ils sont la structure de base pour les bases de données.

Série D'exercices N5

Exercice 1
Ecrire l'algorithme (*programme*) qui permet de créer un fichier qui contient des nombres entiers, de les additionner et d'afficher cette somme.

Exercice 2
Ecrire l'algorithme (*programme*) qui permet de créer un fichier étudiant contenant : le matricule, le nom, le prénom, niveau d'étude, l'âge et la moyenne.

Exercice 3
Ecrire l'algorithme (*programme*) qui permet d'ouvrir le fichier précédent (*étudiant*) et de lui ajouter deux autres enregistrements.

Exercice 4
Ecrire l'algorithme (*programme*) qui permet d'ouvrir le fichier précédent (*étudiant*) et de lui appliquer les requêtes suivantes :

1. Rechercher un étudiant par son matricule
2. Rechercher un étudiant par son nom
3. Rechercher et afficher les noms des étudiants dont leurs moyennes est supérieur ou égales à 10.
4. Rechercher les étudiants dont leurs noms commencent par A.
5. Calculer la moyenne de tous les étudiants de niveau 2.
6. Supprimer l'enregistrement correspond à l'étudiant x donné.

Exercice 5
Ecrire l'algorithme (*programme*) qui permet de trier le fichier précédent (*étudiant*) suivant la moyenne.

Exercice 6
Ecrire l'algorithme (*programme*) qui permet de fusionner deux fichiers et de calculer le nombre d'enregistrements du fichier résultat.

Chapitre 6 :

La Récursivité

Introduction

La récursivité est un concept général qui peut être illustré dans (*quasiment*) tous les langages de programmation, et qui peut être utile dans de nombreuses situations.

1. Définition : la programmation récursive est une technique de programmation qui remplace les instructions de boucles (*while, for, etc.*) par des appels de fonction.

L'idée est de diviser un problème P en sous-problèmes (*certains de la même forme*), de résoudre ces sous-problèmes de manière directe si c'est possible sinon de la même manière que P (*le problème initial*), puis de combiner le résultat de ces sous-problèmes pour résoudre le problème initial P. Autrement dit, on traite le problème P en le ramenant à un problème de la même forme mais plus petit.

2. Exemple

n!= n (n-1) ! et comme il faut s'arrêter, il faut définir une condition d'arrêt, on pose donc 0 !=1.

Le principe de l'exécution est simple : l'algorithme met de côté (*en fait dans une PILE*) ce qu'il ne sait pas faire quitte à les reprendre après. Par conséquent, on doit dire à l'algorithme comment faire dans les cas les plus simples.

Les conditions d'arrêt : ces cas non récursifs sont appelés cas de base et les conditions que doivent satisfaire les données dans ces cas de base sont appelées conditions d'arrêt ou de terminaison. Dans les cas complexes, l'algorithme va faire appel à lui-même en simplifiant ces cas jusqu'à tomber sur l'une des conditions d'arrêt.

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Souvent ce sont les cas les plus simples. Sans cela, on risque de tourner en rond et de faire des exécutions qui ne se finissent pas.

Si ces principes ne sont pas appliqués, l'algorithme risque soit de ne pas produire de résultat, soit de tourner à l'infini

On doit conduire le programme vers les cas de base : tout appel récursif doit se faire avec des données plus proches des conditions de terminaison.

Théoriquement, on peut toujours *dé-récursiver* un algorithme récursif, c'est-à-dire le transformer en algorithme itératif. En pratique, ce n'est pas toujours facile!

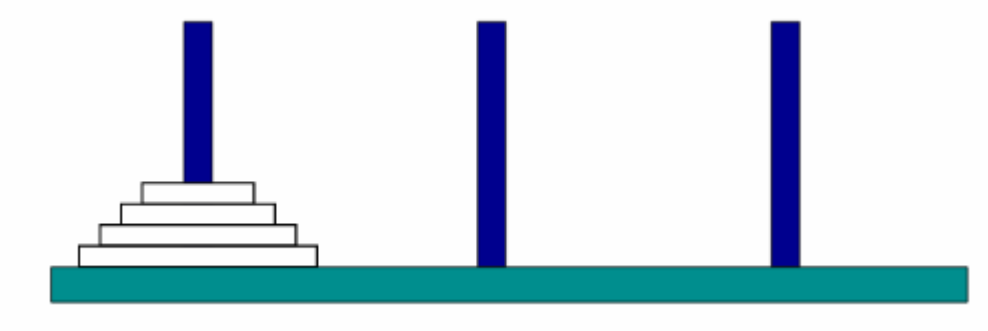
3. Exemple de raisonnement récursif : les tours de Hanoi

Un exemple très connu de raisonnement récursif apparaît dans le problème des tours de Hanoi. Il s'agit de *N* disques de tailles différentes, trouées au centre, qui peuvent être empilés sur trois piliers. Au début, tous les disques sont empilés sur le pilier de gauche, en ordre croissant de la taille, comme dans la figure suivante. Le but du jeu (*exemple*) est de déplacer

les disques un par un pour reconstituer la tour initiale sur le pilier de droite. Il y a deux règles à respecter:

1. On peut seulement déplacer un disque qui se trouve au sommet d'une pile (*non couvert par un autre disque*) ;
2. Un disque ne doit jamais être placé sur un autre plus petit.

Le jeu est illustré dans la figure ci-dessous



La solution s'exprime très facilement par récursivité. On veut déplacer une tour de disques du pilier de gauche vers le pilier de droite, en utilisant le pilier du milieu comme position intermédiaire.

1. Il faut d'abord déplacer vers le pilier du milieu la tour de *n-1* disques du dessus du pilier de gauche (*en utilisant le pilier de droite comme intermédiaire*).
2. Il reste sur le pilier de gauche seul le grand disque à la base. On déplace ce disque sur le pilier de droite.
3. Enfin, on déplace la tour de *n -1* disques du pilier du milieu vers le pilier de droite, au-dessus du grand disque déjà placé (*en utilisant le pilier de gauche comme intermédiaire*).

La fonction récursive est donnée comme suit :

Fonction Hanoi (nb_disques, départ, destination, intermédiaire)
Début
    si (*nb_disques = 1*) **alors**
    déplacer le disque 1 de départ vers la destination
    sinon
    Hanoi (nb_disques -1, départ, intermédiaire, destination)
    déplacer le disque nb_disques de départ vers destination
    Hanoi (nb_disques -1, intermédiaire, destination, départ)
    fsi
Fin.

L'exécution d'un programme qui utilise la fonction Hanoi avec nombre de disque = 3 produit le résultat suivant :

Combien de disques ? **3**

Déplacer le disque de gauche - droite
Déplacer le disque de v gauche - milieu
Déplacer le disque de droit - milieu
Déplacer le disque de gauche - droite
Déplacer le disque de milieu - gauche
Déplacer le disque de milieu - droite
Déplacer le disque de gauche - droite

4. Exemple de raisonnement récursif : La suite de Fibonacci

Les nombres de la suite de Fibonacci sont égaux à la somme des deux termes précédents, ils suivent donc la formule de récurrence :

$$F(n+2)=F(n)+F(n+1)$$

Pour initialiser la suite, on prend généralement F(0) = 0 et F(1) = 1

Calcul récursif de Fibonacci
fib(4) -> fib(3) + fib(2)
-> (fib(2) + fib(1)) + fib(2)
-> ((fib (1) + fib (1)) + fib(1)) + fib(2)
-> ((1 + fib(1)) + fib(1)) + fib(2)
-> ((1 + 1) + fib(1)) + fib(2)
-> (2 + fib(1)) + fib(2)
-> (2 + 1) + fib(2)
-> 3 + fib(2)
-> 3 + (fib(1) + fib(1))
-> 3 + (1 + fib(1))
-> 3 + (1 + 1)
-> 3 + 2
-> 5

Conclusion

Certains problèmes peuvent être résolus plus logiquement en utilisant la récursivité. Les programmes sont plus compacts, plus faciles à écrire et à comprendre. Son usage est naturel quand le problème à traiter peut se décomposer en deux ou plusieurs sous-problèmes identiques au problème initial mais avec des valeurs de paramètres différentes. Refuser la récursivité dans ce dernier cas oblige l'utilisateur à gérer lui-même une pile des différentes valeurs des variables, ce que le système fait automatiquement lors de l'utilisation de la récursivité.

Série D'exercices N6

Exercice 1
Ecrire un algorithme récursif qui permet d'afficher tous les entiers pairs inferieurs à N par ordre décroissant. Ecrire également l'algorithme itératif équivalent.

Exercice 2
Ecrire une fonction récursive qui permet de calculer N!

Exercice 3
Ecrire une fonction récursive qui permet de calculer C_n^p

Exercice 4
Ecrire l'algorithme récursif de tours de Hanoi

Exercice 5
Ecrire un algorithme récursif qui permet de faire la recherche dichotomique dans un tableau trié à une dimension.

Exercice 6
Ecrire un algorithme récursif qui permet de calculer le nième terme de la suite de fibonacci

Chapitre 7 :

La Complexité Algorithmique

Introduction

On n'exige pas seulement d'un algorithme qu'il résolve un problème, on veut également qu'il soit efficace, c'est-à-dire rapide (*en termes de temps d'exécution*) ; peu gourmand en ressources (*espace de stockage, mémoire utilisée*). On a donc besoin d'outils qui nous permettre d'évaluer la qualité théorique des algorithmes proposés. Pour toutes ces raisons, l'étude de la complexité algorithmique est primordiale.

La théorie de la complexité (*algorithmique*) vise à répondre à ces besoins ;

1. de classer les problèmes selon leur difficultés ;
2. de classer les algorithmes selon leur efficacités ;
3. de comparer les algorithmes résolvant un problème donné avant de faire un choix et sans les implémenter ;

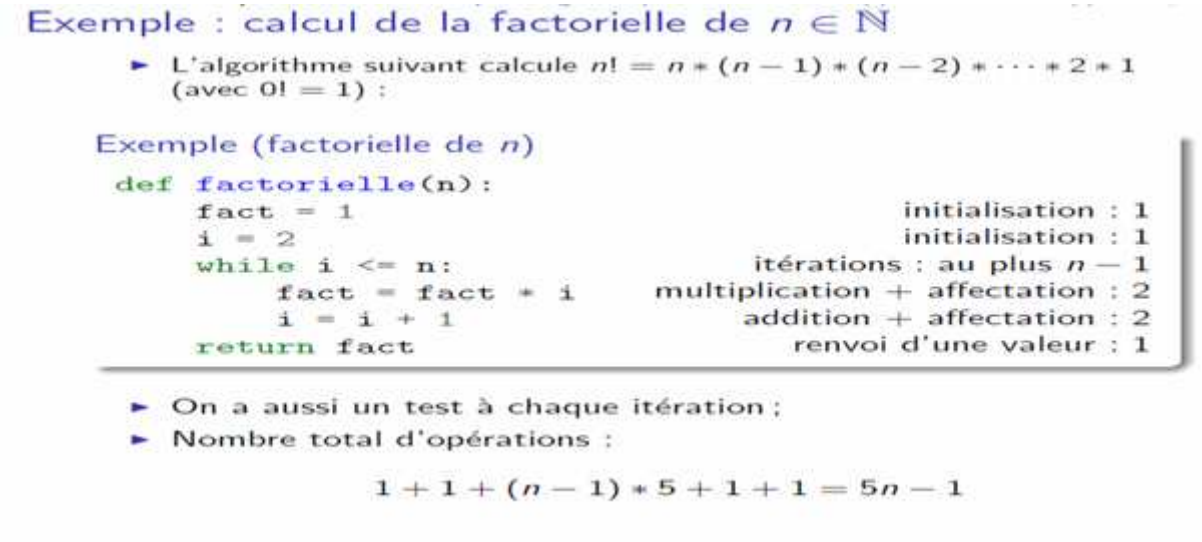
Pour cela, on utilise des unités de temps abstraites et proportionnelles au nombre d'opérations effectuées. Au besoin, on pourra adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute ;

La notation la plus utilisée pour exprimer la complexité d'un algorithme est la notation *O* (*pour ordre de...*), qui dénote un ordre de grandeur. Par exemple, on dira d'un algorithme qu'il est *O*(15) s'il nécessite au plus 15 opérations (*dans le pire cas*) pour se compléter. En langage naturel, on dira qu'il est *O* de 15 ou encore *de l'ordre de* 15.

Pour calculer cette complexité, on applique **les règles suivantes** :

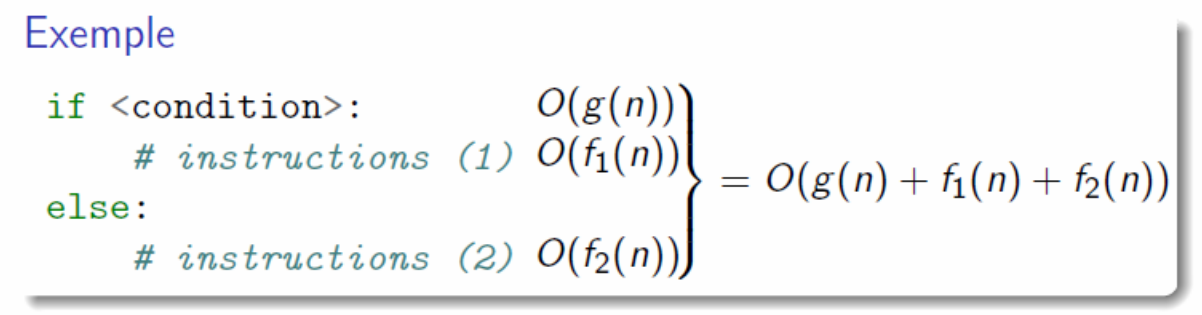
1. Chaque instruction basique consomme une unité de temps ; *(affectation d'une variable, comparaison,* +, -, *, =, ...). Les instructions de base prennent un temps constant, notée O(1)
2. Chaque itération d'une boucle rajoute le nombre d'unités de temps consommées dans le corps de cette boucle ;
3. Chaque appel de fonction rajoute le nombre d'unités de temps consommées dans cette fonction ;
4. On additionne les complexités d'opérations en séquence : O(f1(n)) + O(f2(n)) = O(f1(n) + f2(n)). C'est la même chose pour les branchements conditionnels :
5. Pour avoir le nombre d'opérations effectuées par l'algorithme, on additionne le tout ;

L’exemple suivant illustre comment calculer la complexité de la fonction factorielle.

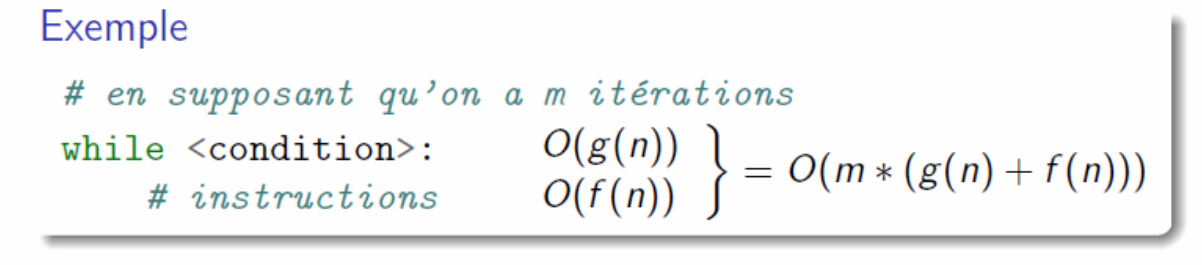


La complexité d'un algorithme est une mesure de sa performance asymptotique (*des données très grandes*) dans le pire cas ;

L’exemple suivant illustre comment calculer la complexité dans l’alternative **if-else**.



Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations. Par exemple, la complexité d'une boucle **while** se calcule comme suit :



La notation O, dite notation de Landau, vérifie **les propriétés suivantes** :
- Si f=O(g) et g=O(h) alors f=O(h)
- si f=O(g) et k un nombre, alors k*f=O(g)
- si f1=O(g1) et f2=O(g2) alors f1+f2 = O(g2+g2)
- si f1=O(g1) et f2=O(g2) alors f1*f2 = O(g2*g2)

Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque partie de l'algorithme ;
2. on combine ces complexités conformément aux règles qu'on vient de voir ;
3. on simplifie le résultat grâce aux règles de simplification (*élimination des constantes, et conservation du (des) terme(s) dominant(s)*)

-Quelques équivalences

- f (n) = O(g(n)) n'implique pas g(n) = O(f (n)) ,
contre-exemple : $5n + 43 = O(n^2)$, mais $n^2 \neq O(n)$;
- f (n) ≠ O(g(n)) n'implique pas g(n) ≠ O(f (n)) :
contre-exemple : $18\ n^3 - 35\ n \neq O(n)$, mais $n = O(n^3)$;
- On dit que deux fonctions f(n) et g (n) sont équivalentes si f(n)= O (g (n)) et g(n) = o (f (n))

D'un point de vue algorithmique, trouver un nouvel algorithme de même complexité pour un problème donné ne présente donc pas beaucoup d'intérêt ;

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

1. La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.
2. La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (*en pratique, cette complexité n'est pas très utile*).
3. La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution. Le tableau suivant illustre les principales classes de complexité.

Complexité	Classe
$O(1)$	constant
$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	sous-quadratique
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel

Conclusion

L'analyse de la complexité des algorithmes étudie formellement la *quantité de ressources* en temps et en espace nécessitée par l'exécution d'un algorithme donnée. La complexité d'un algorithme est une mesure du temps requis par l'algorithme pour accomplir sa tâche, en fonction de la taille de l'échantillon à traiter.

Série D'exercices N7

Exercice 1
Donner la complexité du code suivant

```
T :=0 ;
Pour i allant de 1 n faire
Début
        Pour j allant de 1 à m faire
        Début
        T :=T+1 ;
        Fin ;
Fin ;
```

Exercice 2
Donner la complexité du code suivant

```
T :=0 ;
Pour i allant de 1 n faire
Début
        Pour j allant de i à m faire
        Début
        T :=T+1 ;
        Fin ;
Fin ;
```

Exercice 3
Donner la complexité du code suivant

```
i :=1 ;
a :=1 ;
Tant que (a<10) faire
début
i :=i+1 ;
a :=i/2 ;
fin ;
```

Exercice 4
Donner la complexité du code suivant

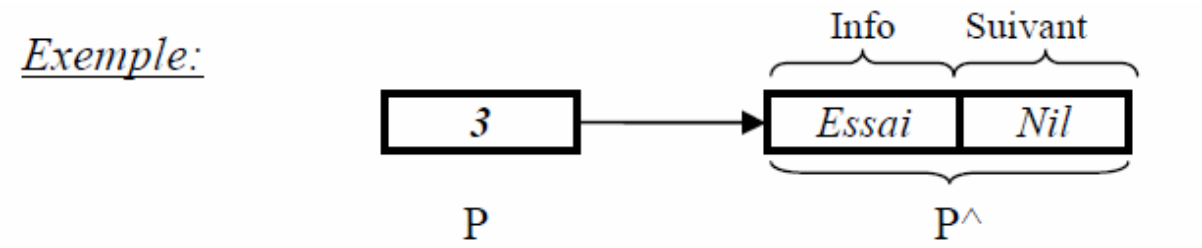
```
Si (a<10) alors
P := fact(a) ;
Sinon
P := fact(a/2);
```

Chapitre 8 :

Les Pointeurs

1. Introduction

Un **pointeur** est une variable dont la valeur est une adresse mémoire. Un pointeur, noté *P*, pointe sur une variable dynamique notée *P^*.
Le **type de base** est le type de la variable pointée. Le **type du pointeur** est l'ensemble des adresses des variables pointées du type de base. Il est représenté par le symbole ^ suivi de l'identificateur du type de base.

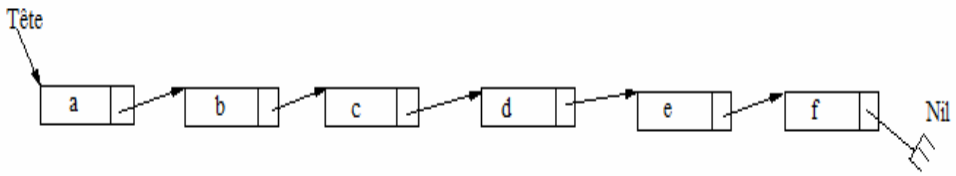


La variable pointeur *P* pointe sur l'espace mémoire *P^* d'adresse 3. Cette cellule mémoire contient la valeur "*Essai*" dans le champ *Info* et la valeur spéciale *Nil* dans le champ *Suivant*. Ce champ servira à indiquer quel est l'élément suivant lorsque la cellule fera partie d'une liste. La valeur *Nil* indique qu'il n'y a pas d'élément suivant. ***P^*** est l'objet dont l'adresse est rangée dans *P*.

Avec les pointeurs, plusieurs structures de données peuvent être définies : les listes linéaires chainées, les files d'attente, les piles, les arbres.

2. Liste Linéaire Chainée

Une liste linéaire chaînée (LLC) est un ensemble de maillons, alloués dynamiquement, chaînés entre eux. Schématiquement, on peut la représenter comme suit : abcdef



Un élément ou maillon d'une LLC est toujours une structure (*Objet composé*) avec deux champs : un champ *Valeur* contenant l'information et un champ *Adresse* donnant l'adresse du prochain élément. A chaque maillon est associée une adresse. On utilise ainsi une nouvelle classe d'objet : le type *Pointeur* qui est une variable contenant l'adresse d'un emplacement mémoire.

Les listes chaînées entraînent l'utilisation de procédures d'allocation et de libération dynamiques de la mémoire. Ces procédures sont les suivantes:

Allouer(P) : réserve un espace mémoire *P^* et donne pour valeur à *P* l'adresse de cet espace mémoire. On alloue un espace mémoire pour un élément sur lequel pointe *P*.
Désallouer(P) : libère l'espace mémoire qui était occupé par l'élément à supprimer *P^* sur lequel pointe P.

Pour définir les variables utilisées dans l'exemple ci-dessus, il faut : définir le type des éléments de la liste :

```
Type Cellule= Structure
        Info : Chaîne
        Suivant : Liste
fin Structure
```

- définir le type du pointeur : Type Liste = ^Cellule
- déclarer une variable pointeur : Var P : Liste
- allouer une cellule mémoire qui réserve un espace en mémoire et donne à P la valeur de l'adresse de l'espace mémoire P^ : Allouer(P)
- affecter des valeurs à l'espace mémoire P^: P^.Info← "Essai" ; P^.Suivant← Nil

Quand *P = Nil* alors *P* ne pointe sur rien

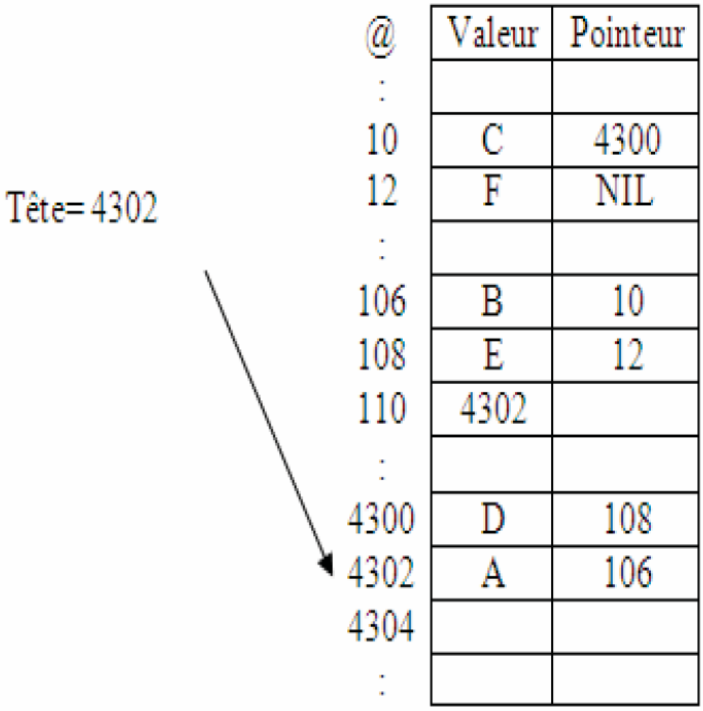
Il faut commencer par définir un type de variable pour chaque élément de la chaîne. En langage algorithmique ceci se fait comme suit :

```
Type Liste = ^Elément
Type Elément = Structure
        Info : variant
        Suivant : Liste
Fin structure
Variables Tète, P : Liste
```

Le type de *Info* dépend des valeurs contenues dans la liste : entier, chaîne de caractères, variant pour un type quelconque…

Une liste linéaire chainée est caractérisée par l'adresse de son premier élément souvent appelé *Tête*. *Nil* constitue l'adresse qui ne pointe aucun maillon, utilisé pour indiquer la fin e la liste dans le dernier maillon

La représentation réelle en mémoire de cette liste ressemble à la représentation suivante :



Dans le langage algorithmique, on définira le type d'un maillon comme suit :

```
Type  TMaillon = Structure
    Valeur : Typeqq ; // désigne un type quelconque
    Suivant : Pointeur(TMaillon) ;
Fin ;
```

Afin de développer des algorithmes sur les listes linéaires chainées, on construit une machine abstraite avec les opérations suivantes : Allouer, Libérer, Aff_Adr, Aff_Val, Suivant et Valeur définies comme suit :

– Allouer(P) : allocation d'un espace de taille spécifiée par le type de P. L'adresse de cet espace est rendue dans la variable P de type Pointeur.
– Libérer(P) : libération de l'espace pointé par P.
– Valeur(P) : consultation du champ *Valeur* du maillon pointé par P.
– Suivant(P) : consultation du champ *Suivant* du maillon pointé par P.
– Aff_Adr(P, Q) : dans le champ *Suivant* du maillon pointé par P, on range l'adresse Q.
– Aff_Val(P,Val) : dans le champ *Valeur* du maillon pointé par P, on range la valeur Val.

Cet ensemble d'opérations (*ou de méthodes ou fonctions*) est appelé modèle.

Les principaux traitements qui peuvent être effectué sur des listes sont les suivants :
- Créer une liste.
- Ajouter un élément.

- Supprimer un élément.
- Modifier un élément.
- Parcourir une liste.
- Rechercher une valeur dans une liste

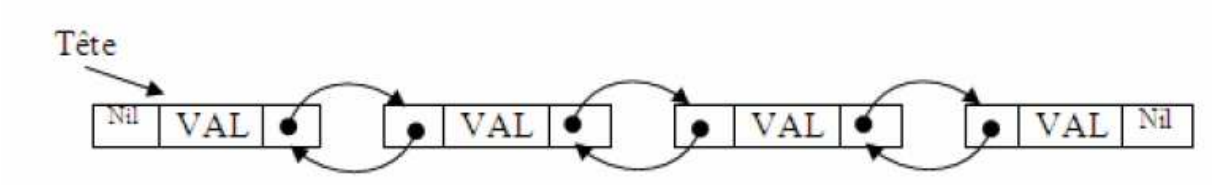
Algorithme CreationLLC avec N éléments et afficher ces valeurs à la fin de traitement.

```
Algorithme CréerListe;
Type  TMaillon = Structure
    Valeur : Typeqq ;
    Suivant : Pointeur(TMaillon) ;
Fin ;
Var  P, Q, Tete : Pointeur(TMaillon ;
    i, Nombre : entier ;
    Val : Typeqq ;
Début
  Tete ← Nil ;
  P ← Nil ;
  Lire(Nombre) ;
```

```
Pour i de 1 à Nombre faire
    Lire(Val) ;
    Allouer(Q) ;
    Aff_val(Q, val) ;
    Aff_adr(Q, NIL) ;
    Si (Tete ≠ Nil) Alors
        Aff_adr(P, Q)
    Sinon
        Tete ← Q
    Fin Si;
    P ← Q ;
Fin Pour;
P ← Tete ;
Tant que ( P ≠ Nil) faire
    Ecrire(Valeur(P)) ;
    P ← Suivant(P) ;
Fin TQ;
Fin.
```

Il existe différents types de listes chaînées :

1. **Liste chaînée simple** constituée d'éléments reliés entre eux par des pointeurs.
2. **Liste chaînée ordonnée** où l'élément suivant est plus grand que le précédent. L'insertion et la suppression d'élément se font de façon à ce que la liste reste triée.
3. **Liste circulaire** où le dernier élément pointe sur le premier élément de la liste. S'il s'agit d'une liste doublement chaînée alors de premier élément pointe également sur le dernier.
4. **Listes linéaires chaînées bidirectionnelles :** c'est une LLC où chaque maillon contient à la fois l'adresse de l'élément précédent et l'adresse de l'élément suivant ce qui permet de parcourir la liste dans les deux sens.



Un exemple de déclaration de listes linéaires chainées bidirectionnelles est donné comme suit :

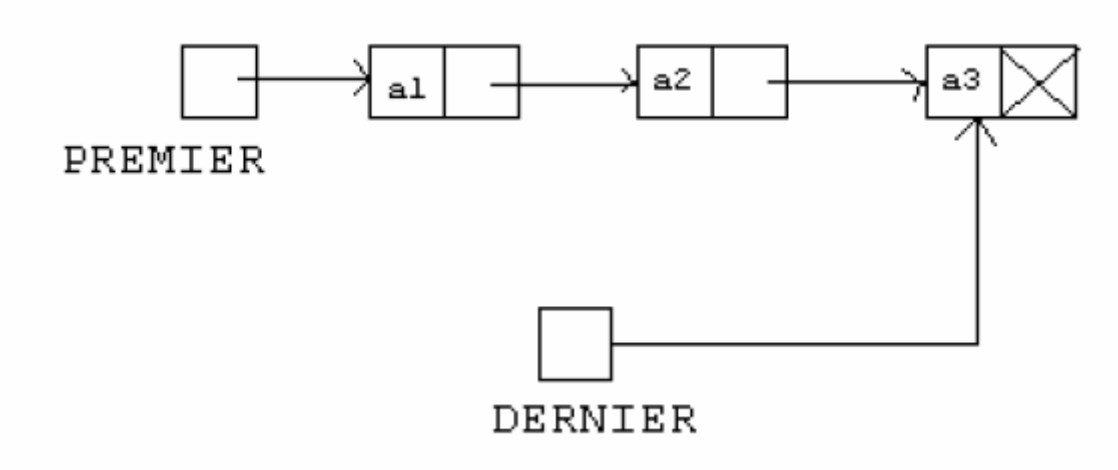
Déclaration

```
Type  TMaillon = Structure
    Valeur : Typeqq ; // désigne un type quelconque
    Précédent, Suivant : Pointeur(TMaillon) ;
Fin ;
Var  Tete : TMaillon ;
```

Un modèle sur les listes linéaires chaînées bidirectionnelles est donné comme suit :
– **Allouer (p):** création d'un maillon.
– **Libérer (p):** libérer un maillon.
– **Valeur (P)** : retourne la valeur du champ *valeur* du maillon pointé par P.
– **Suivant(P)** : retourne le pointeur se trouvant dans le champ *suivant* du maillon pointé par P
– **Précédent(P**) : retourne le pointeur se trouvant dans le champ *précédent* du maillon pointé par P
– **Aff_Val(P,x)** : Ranger la valeur de *x* dans le champs *val*eur du maillon pointé par P
– **Aff_Adr_Précedent(P,Q)** : Ranger *Q* dans le champs *précédent* du maillon pointé par P
– Aff_Adr_Suivant(P,Q) : Ranger *Q* dans le champs *suivant* du maillon pointé par P

3. File

Une file est une structure de données dynamique dans laquelle on insère des nouveaux éléments à la fin (*queue*) et où on enlève des éléments au début (*tête de file*). L'application la plus classique est la file d'attente, et elle sert beaucoup en simulation. Elle est aussi très utilisée aussi bien dans la vie courante que dans les systèmes informatiques. Par exemple, elle modélise la file d'attente des clients devant un guichet, les travaux en attente d'exécution dans un système de traitement par lots, ou encore les messages en attente dans un commutateur de réseau téléphonique. On retrouve également les files d'attente dans les programmes de traitement de transactions telle que les réservations de sièges d'avion.
On note bien que dans une file, le premier élément inséré est aussi le premier retiré. On parle de mode d'accès FIFO (*First In Fist Out*).



Les principales opérations sont :
- **Enfiler** : qui consiste à ajouter un nouveau élément en queue de la file et
- **Défiler** qui consiste à supprimer un élément de la file.

Pour ajouter un nouveau élément, on doit se déplacer en queue de la file en parcourant la file de début jusqu'à la fin. Lorsque le dernier nœud est atteint, lui accrocher le nouvel élément qui devient de ce fait le dernier.

Quatre opérations de base sont principalement utilisées dans une file d'attente:

1. Consulter le premier élément de la file,
2. Tester si la file est vide,
3. Enfiler un nouvel élément: le mettre en dernier,
4. Défiler un élément, le premier (*le supprimer*).

4. Pile

Les piles sont des structures linéaires pour lesquelles à chaque nœud on s'intéresse en premier lieu non seulement à la valeur stockée mais aussi au moment où celle-ci est arrivée. Une pile est un type de liste particulier dans lesquels toute insertion ou suppression d'élément se fait à une extrémité appelée *sommet* de la pile. Pour bien comprendre les mécanismes de base, il suffit de penser à une pile d'assiettes.

- **Empiler** un élément dans la pile consiste à créer un nouveau nœud et à le placer au sommet de la pile, il devient alors lui-même le nouveau sommet de la pile.
- **Dépiler** un élément de la pile consiste à récupérer le sommet de la pile et faire pointer le sommet de la pile à son successeur (*le second élément dans la pile*)

Une pile est une liste sur laquelle on autorise seulement 4 opérations:

1. Consulter le dernier élément de la pile (*le sommet de la pile*),
2. Tester si la pile est vide,
3. Empiler un élément, le mettre au sommet de la pile ==> PUSH,
4. Dépiler un élément (*par le sommet*) ==> POP.

Toutes les opérations sont effectuées sur la même extrémité; on parle de structure en FIFO. Comportement d'une pile: Last In First Out (LIFO).

5. Arbre

L'arbre est une structure de données qui généralise la liste, alors qu'une cellule de la liste a un seul successeur (*leur suivant*), dans un arbre il peut y en avoir plusieurs. On parle alors de **nœud** (*au lieu de cellule*). Un **nœud père** peut avoir plusieurs **nœuds fils**. Un fils n'a qu'un seul père, et tous les nœuds ont un ancêtre commun appelé la **racine** de l'arbre (*le seul nœud qui n'a pas de père*).
Comme les listes, la structure d'arbre est fortement utilisée en informatique, particulièrement en base de données, pour représenter des algorithmes de recherche opérationnelle, en traitement d'images, dans les programmes de calcul formel,

5.1. Définition : un arbre A est un ensemble fini d'éléments tels que :

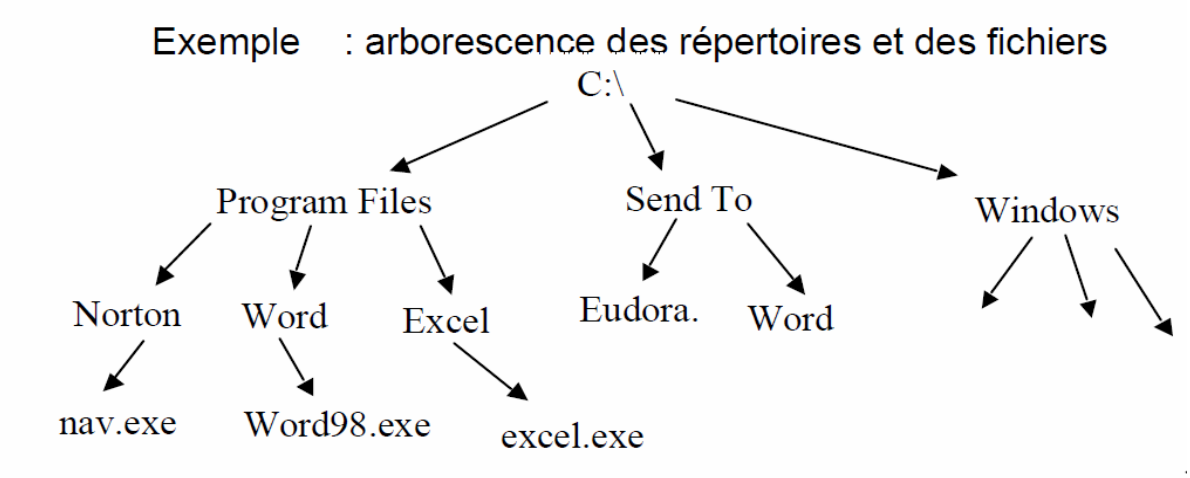
- Il existe un élément particulier appelé racine d'arbre,
- Les autres éléments sont repartis de manière à former une partition en *m* classes (m>=0), A1, A2, Am, chacune de ces classes étant elle-même un arbre (*sous arbre*),

La racine est l'élément unique de l'arbre à partir duquel on a accès à ses éléments. Tous les algorithmes de traitement d'arbres partiront de la racine. Pour s'orienter dans un arbre, on utilise la terminologie classique des arbres généalogiques : père, fils, frères, ascendant, descendant,… tel que :

Père d'un sommet : le prédécesseur d'un sommet
Fils d'un sommet : les successeurs d'un sommet
Frères : des sommets qui ont le même père
Nœud = sommet
Racine : nœud sans père
Feuille : nœud sans fils
Branche : chemin entre deux nœuds

Niveau (*profondeur*) d'un nœud : la longueur de la branche depuis la racine
Hauteur d'un nœud : la longueur de la plus longue branche de ce nœud jusqu'à une feuille
Hauteur d'un arbre : la hauteur de la racine
Taille d'un arbre : nombre de ses sommets

Le **degré** d'un nœud dans un arbre est égale au nombre de fils de ce nœud. Un nœud **feuille** est un nœud de degré 0.



5.2. Parcours d'arbres

Les principales parcourt d'arbres sont :

1. **Parcours en pré-ordre ou Parcours préfixe** (Racine Gauche Droit) **:** au niveau de chaque nœud, on traite la valeur disponible à cet endroit, puis on continue le parcours, cette fois sur le sous-arbre gauche puis sur le sous arbre droit. En d'autres mots : le nœud racine est traité au premier passage avant le parcours des sous-arbres. L'algorithme associé est donné comme suit.

```
ParcoursRGD (ARBIN A)
Début
    Si (A est vide) Alors
            terminaison (A)
    Sinon
            traitement (A)
            Parcours (ABG de A)
            Parcours (ABD de A)
    Fsi
Fin
```

2. **Parcours symétrique ou infixe** (Gauche Racine Droit)**:** cette fois-ci, le traitement se fait entre les deux parcours, du sous arbre droit et du sous arbre gauche. En d'autres mots : le nœud racine est traité au second passage après le parcours du sous-arbre gauche et avant le parcours du sous-arbre droit.

```
ParcoursGRD (ARBIN A)
Début
    Si (A est vide) lors
            Terminaison (A)
    Sinon
            Parcours (ABG de A)
            Traitement (A)
            Parcours (ABD de A)
    Fsi
Fin
```

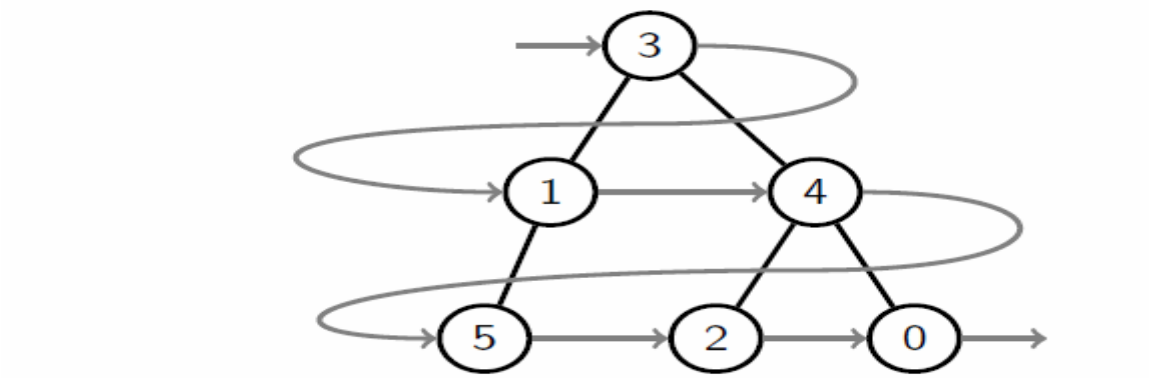
3. **Parcours terminal ou post fixe ou suffixe** (Gauche Droit Racine)**:** on commence à parcourir les deux sous arbres de nœud courant puis ensuite effectuer le traitement proprement dit. En d'autres mots : le nœud racine est traité au dernier passage après le parcours des sous-arbres

```
ParcoursGDR (ARBIN A)
Début
        Si (A est vide) Alors
                terminaison (A)
        Sinon
                Parcours (ABG de A)
                Parcours (ABD de A)
                Traitement (A)
        Fsi
Fin
```

D'autres types de parcourt d'arbres sont donnés comme suit :

- en largeur.
- en profondeur:

Parcours en largeur d'abord.
On parcourt par distance croissante à la racine. La figure suivante illustre son principe.



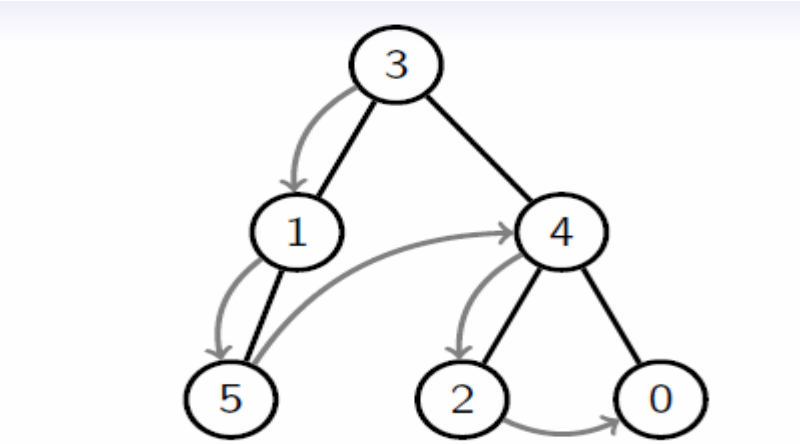
Si le traitement d'un sommet consiste à l'afficher, on affichera dans l'ordre 3, 1, 4, 5, 2, 0.
L'algorithme associé est donné comme suit :

```
Parcours EnLargeur (ARBIN A)
Début
        créer une file vide F
Si (A est non vide) Alors
        Enfiler A dans F
TQ (F non vide) Faire
        A ← Défiler (F)
        traitement (A)
Si (ABG de A non vide) Alors
        Enfiler ABG de A dans F
Fsi
Si (ABD de A non vide) Alors
        Enfiler ABD de A dans F
Fsi
Ftq
Fsi
```

détruire F
Fin

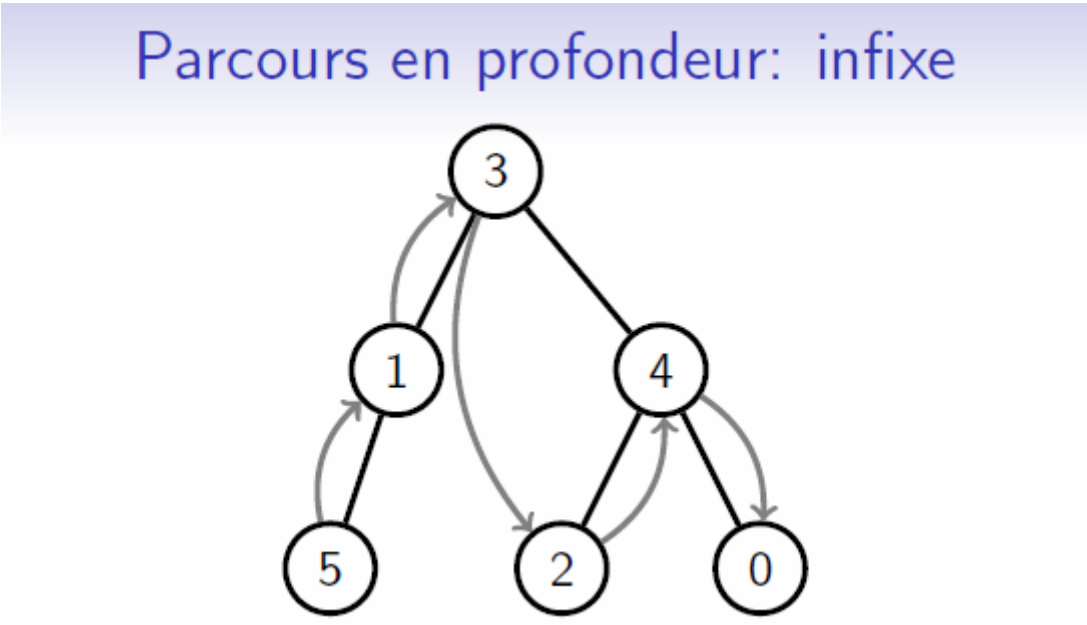
Parcours en profondeur d'abord
Deux cas peuvent se présenté : infixe et postfixe

La figure suivante illustre son principe.



Si le traitement d'un sommet consiste à l'afficher, on affichera dans l'ordre 3, 1, 5, 4, 2, 0.
Le programme associé est donné comme suit :

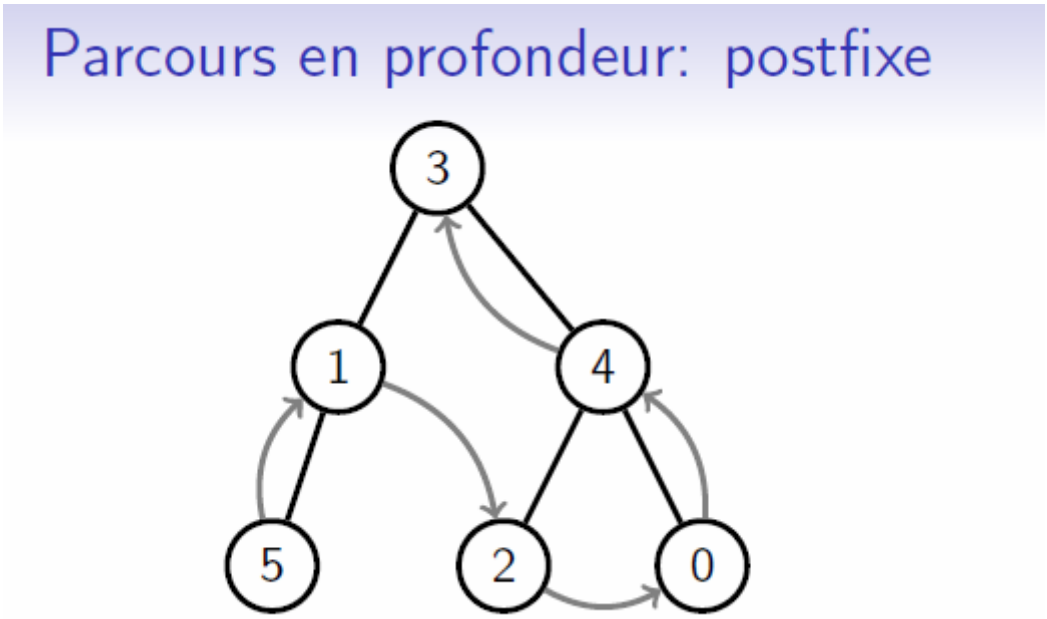
```
static void Affiche(Arbre a) {
i f (a==null) return;
System.out.println(a.val+" ");
Affiche(a.gauche);
Affiche(a.droite);
}
```



Si le traitement d'un sommet consiste à l'afficher, on affichera dans l'ordre 5, 1, 3, 2, 4, 0.

Le programme associé est donné comme suit :

```
static void Affiche(Arbre a) {
i f (a==null) return;
Affiche(a.gauche);
System.out.println(a.val+" ");
Affiche(a.droite);
}
```



Si le traitement d'un sommet consiste à l'afficher, on affichera dans l'ordre 5, 1, 2, 0, 4, 3.
Le programme associé est donné comme suit :

```
static void Affiche(Arbre a) {
i f (a==null) return;
Affiche(a.gauche);
Affiche(a.droite);
System.out.println(a.val+" ");
}
```

Ecriture des expressions

Ecriture infixée

Le signe d'opération est entre ses arguments

((3 + 12) * 5 + (4 + 17) * 6) / (7 + (8 * 4))

Ecriture postfixée

Le signe d'opération est après ses arguments

4 12 + 5 .* 4 17 + 6 * + 7 8 4 * + /

Ecriture préfixée

Le signe d'opération est avant ses arguments

/ + * + 3 12 5 * + 4 17 6 + 7 * 8 4

6. Conclusion

Certain langages ne permettent pas l'utilisation explicite de pointeurs. La majorité des langages de programmation utilisent (*ou au moins permettent*) le passage de paramètres par valeur. L'utilisation des pointeurs est très puissante dans certains langages. Cette puissance (*et surtout le fait que l'on touche directement à la mémoire sans aucun contrôle*) complexifie le développement d'une application.

Si l'on ne fait pas attention et que l'on accède à une zone mémoire qui ne nous est pas allouée, le processeur via le système d'exploitation génèrera une erreur de segmentation qui provoquera une exception voire fera planter l'application. De plus, comme les allocations mémoire sont réalisées en partie par le développeur, il doit également se charger de la libération de la mémoire lorsqu'il n'en a plus besoin, au risque de voir une fuite mémoire apparaître.

Série D'exercices N8

Exercice 1

Ecrire les algorithmes de base suivants sur les listes linéaires chainées unidirectionnelles :

1. Suppression par valeur,
2. Suppression par position,
3. Inversement d'une liste,

-En créant une nouvelle liste

-Sans créer une nouvelle liste (*En inversant le chainage des mêmes maillons*)

4. Tri par la méthode "sélection et permutation",
5. Tri par la méthode des bulles,
6. Recherche de l'élément qui a le plus grand nombre d'occurrences

Ecrire les algorithmes de base suivants sue les listes linéaires chainées bidirectionnelles :

1. Recherche d'un élément,
2. Suppression d'un élément,
3. Inversement d'une liste

Exercice 2

Soient L1 et L2 deux listes linéaires chainées unidirectionnelle. Ecrire la procédure qui permet de construire la liste L = L1 - L2 contenant tous les éléments appartenant à L1 et n'appartenant pas à L2.

Exercice 3

Ecrire les sous-algorithmes suivants :

1. Suppression de l'ième élément ;
2. Recherche d'un élément ;
3. Concaténer deux piles ;
4. Eclater une pile en deux : l'une pour les éléments paire, et l'autre pour les éléments impaire.

Exercice 4

Concevoir deux sous-algorithmes, qui créent respectivement :

- La file inverse d'une pile ;
- La pile inverse d'une file.

Exercice 5

Ecrire les sous algorithmes suivants :

1. création d'un arbre binaire,
2. recherche d'un élément dans un arbre
3. ajout d'un élément dans un arbre binaire
4. suppression d'un élément dans un arbre binaire

Références Bibliographiques

[1] : Jacques Courtin, *Initiation à l'algorithmique et aux structures de données*, Edition DUNOD, 1998

[2] : Mc Belaid, *Algorithmique et Structures de données*, Edition les pages bleus

[3] : Michel Divay, *Algorithmes et structures de données génériques* - 2ème édition, Edition Dunod

[4] : Zegour Djamel eddine, *Structures de données et de fichiers. Programmation Pascal et C*, édition CHIHAB,

[5] : Thabet Slimani, *Programmation et structures de données avancées en langage C, cours et exercices corrigés*, 2014,

[6] : Claude Pair, Marie-Claude Gaudel, *Les Structures de données et leur représentation en mémoire*, édition Iria, 1979