Algorithmique 2

Pr. MOHAMED CHAKRAOUI

INTRODUCTION À L'ALGORITHMIQUE

- une suite d'étapes dont le but est de décrire la résolution d'un problème ou l'accomplissement d'une tâche.
- une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer dans un ordre bien défini.
- Procédé permettant de résoudre un problème en un nombre fini d'opérations.
- une suite finie et non ambigüe d'opérations permettant de résoudre un problème.

- Algorithme sur machine:
  - *Objectif: donner à la machine la possibilité de **substituer** l'homme pour accomplir une tâche dans un **temps relatif bien réduit**.*
  - *Comment ?*
    - Décrire (analyser et concevoir) la tâche d'une manière compréhensible par la machine.
    - Décomposer la tâche en opérations élémentaires.
    - Traduire l'algorithme en un langage de programmation
    - Programme.

INTRODUCTION À L'ALGORITHMIQUE

- Pour fonctionner, **un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter**

- **la vérification méthodique, pas à pas, de chacun de vos algorithmes représente plus de la moitié du travail à accomplir... et le gage de vos progrès.**

CONVENTIONS D'ÉCRITURE

- une représentation graphique, avec des carrés, des losanges, etc. qu'on appelait des **organigrammes**

- «**pseudo-code**» ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe.

Objectif Ultime du cours

Algorithme

- Terminaison ?
- Produit résultat attendu ?
- Réalisable ? (complexité)
- Peut on faire mieux ? (optimalité)

Programme

- Langage de Programmation
- Ordinateur

Algorithme

- Pseudo code
- Model of computation: mesurer complexité d'un algo en temps d'execution et en espace mémoire

QUESTION

- Q: Utilité de l'algorithmique ?!
- R: exemples
- – Internet: Recherche de routes optimales pour l'acheminement des données.
- Commerce électronique: La cryptographie s'appuient sur des algorithmes numériques pour préserver la confidentialité.
- Une compagnie pétrolière veut savoir où placer ses puits de façon à maximiser les profits.
- – …

ANALYSE ET CONCEPTION DES ALGORITHMES

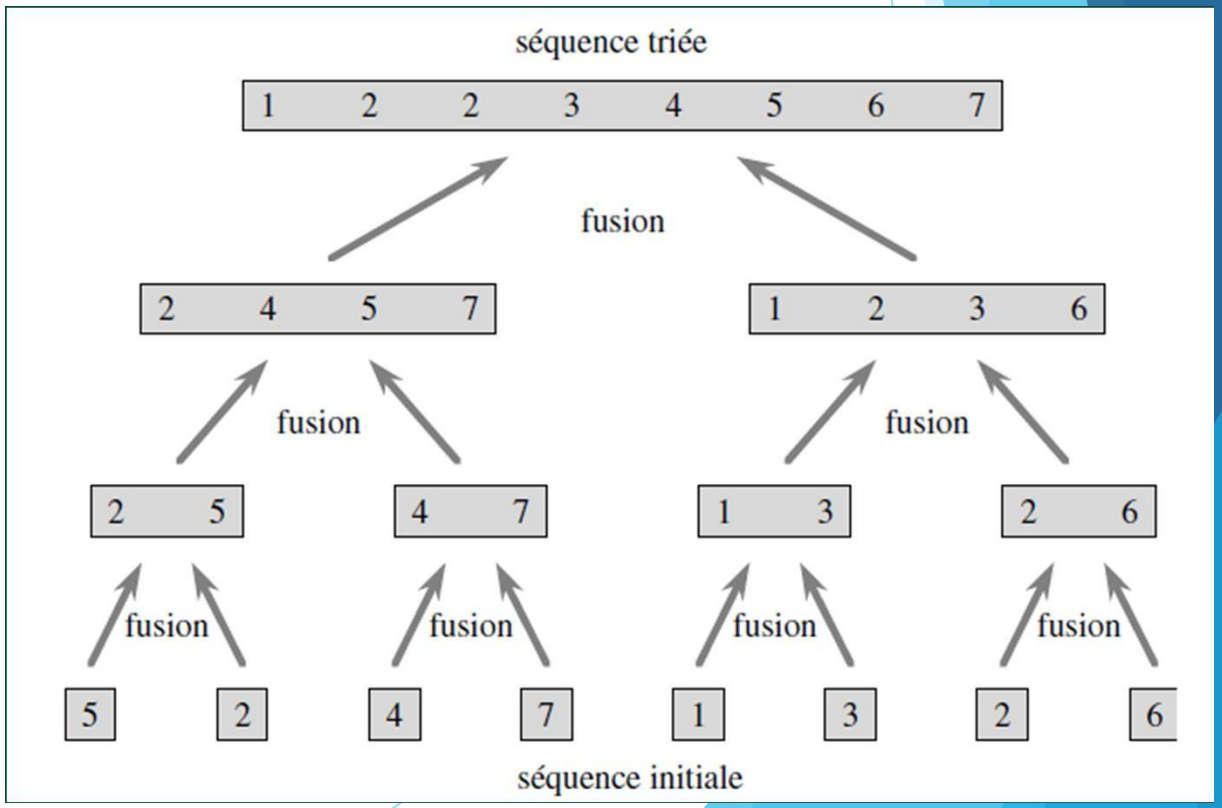
- Analyser un algorithme :
  - Prévoir les ressources nécessaires à cet algorithme.
    - la mémoire,
    - le processeur,
    - mais, le plus souvent, c'est le temps de calcul qui nous intéresse.
- Plusieurs algorithmes peuvent en résulter
  - éliminer les algorithmes inférieurs et garder la meilleure solution.
- Analyse du meilleur cas, du plus défavorable, ou du cas moyen

ANALYSE ET CONCEPTION DES ALGORITHMES

- Conception:
  - **Méthode incrémentale**: utilise un processus **itératif** où chaque itération augmente la quantité d'information.
    - Ex: Tri par insertion (cf. plus loin)
- Diviser pour régner: Analyse Descendante : Décomposer chaque Problème "Composé" en sous Pb (Décomposable ou pas), jusqu'à l'arrivé à des sous Pb élémentaires. (feuille de l'arbre)
- Analyse Ascendante: Reconstitution de la solution de chaque Pb en regroupant les solutions de ses sous Pb. (idem pour le temps d'execution)
  - Ex: Tri par fusion (cf. plus loin)

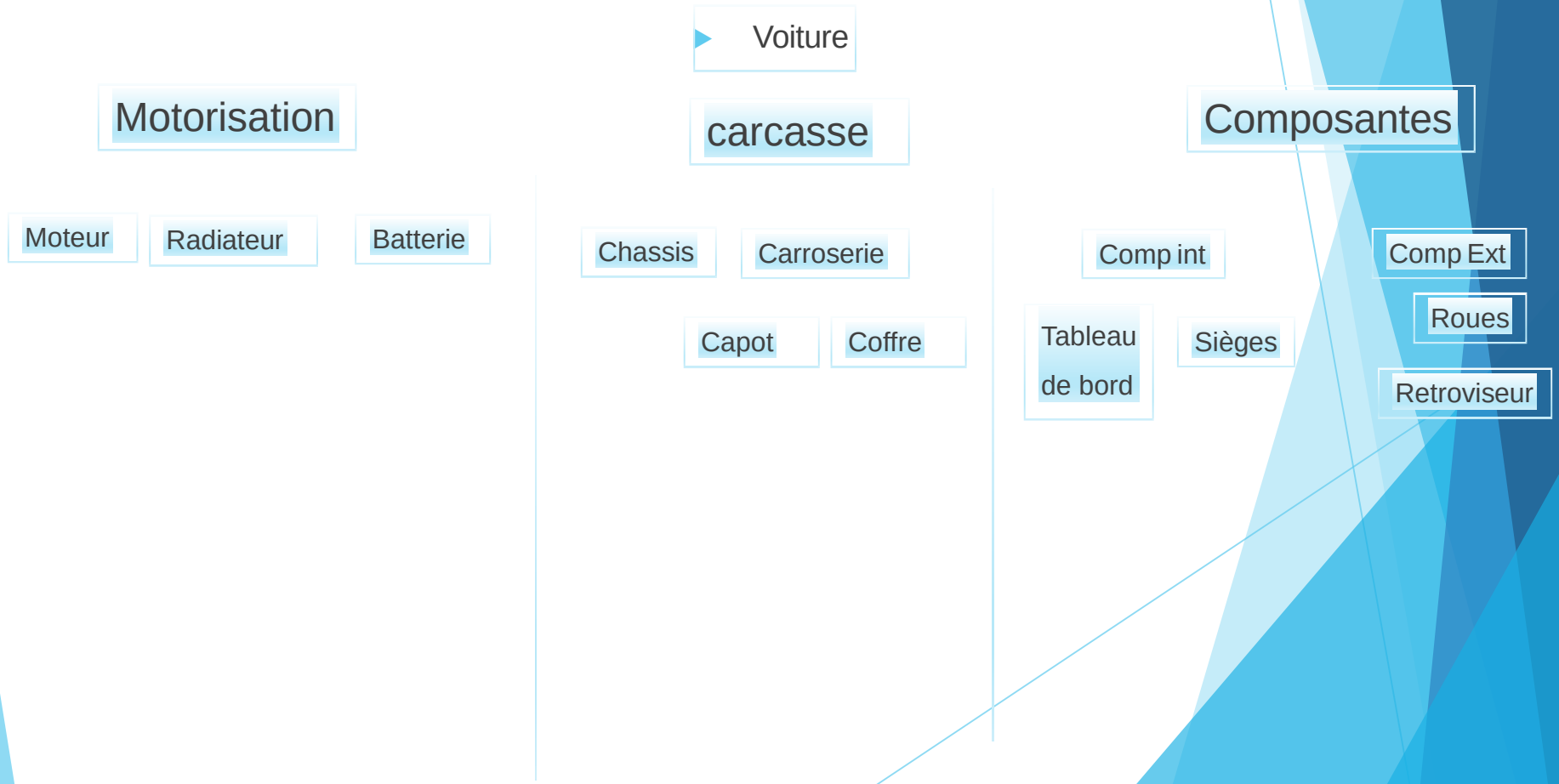
ANALYSE ET CONCEPTION DES ALGORITHMES

- Analyse ascendante :
  - À chaque phase finale (indissociable -indivisible) on passe à la réalisation.
  - Le résultat est fournis à la phase d'avant (niveau plus haut dans l'arborescence)
  - Ex: Tri par Fusion



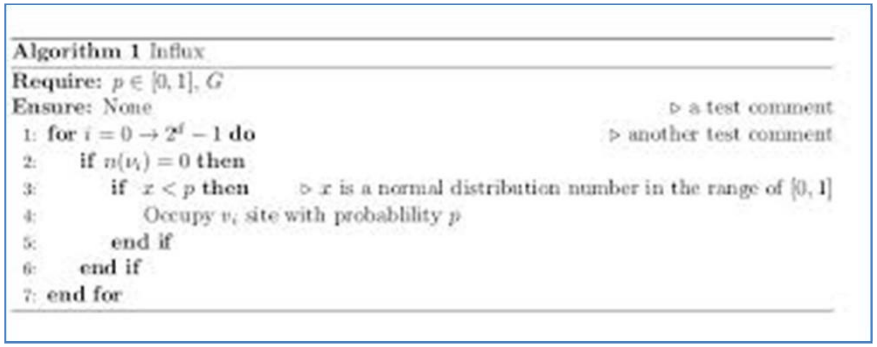
ANALYSE ET CONCEPTION DES ALGORITHMES

- Exemple approximatif: construction d'une voiture



Pseudo code

- Corps: contient
  - Mot clé **Début (Begin)**
  - Une suite d'**instructions**
  - Mot clé **Fin (End)**
- Indentation (style): ajout tabulations et espaces pour une **bonne lisibilité du code**



Algorithm 1 Influx

Require: $p \in [0, 1]$, G
Ensure: None ▷ a test comment

```
for i = 0 → 2^f − 1 do                ▷ another test comment
    if n(v_i) = 0 then
        if x < p then                 ▷ x is a normal distribution number in the range of [0, 1]
            Occupy v_i site with probablility p
        end if
    end if
end for
```

Code Indenté

```
if(!Array.forEach){Array.prototype.forEach=function(D,E){var
C=E||window;for(var
B=0,A=this.length;B<A;++B){D.call(C,this[B],B,this)}};Array.prototy
pe.map=function(E,F){var D=F||window;var A=[];for(var
C=0,B=this.length;C<B;++C){A.push(E.call(D,this[C],C,this))}return
A};Array.prototype.filter=function(E,F){var D=F||window;var
A=[];for(var
C=0,B=this.length;C<B;++C){if(!E.call(D,this[C],C,this)){continue}
A.push(this[C])}return A};Array.prototype.every=function(D,E){var
C=E||window;for(var
B=0,A=this.length;B<A;++B){if(!D.call(C,this[B],B,this)){return
false}}return true};Array.prototype.indexOf=function(B,C){var
C=C||0;for(var A=0;A<this.length;++A){if(this[A]===B){return
A}}return -
1}}Array.prototype.contains=function(A){if(Array.contains){return
this.contains(A)}return this.indexOf(A)>-
1};Array.prototype.insert=function(A){if(!this.contains(A)){this.p
ush(A)}};if(!Array.remove){Array.remove=function(D,C,B){var
A=D.slice((B||C)+1||D.length);D.length=C<0?D.length+C:C;return
D.push.apply(D,A)}}Function.prototype.method=function(A,B){this.pr
```

Code non Indenté

PSEUDO CODE

- Exemple d'algorithme écrit en pseudo-code

```
Nom       : Le nom de l'algorithme
Role      : que fait l'algorithme
Entrée    : les données nécessaires
Sortie    : les résultats produits par l'algorithme
```

Facultatifs

```
Variables : Déclarations des variables
Début
    Instruction 1
    Instruction 2
    …
    Instruction n
Fin
```

/* utiliser les commentaires comme aide mémoire */

Les Variables

- Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs de types différents: utilisation de variables
- 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.
- Outils d'identification de l'information
- Déclaration: nom + type de la variable
  - Nom:
    - Comporte des lettres et des chiffres,
    - Ne comporte pas des signes de ponctuation, en particulier les espaces.
    - Doit commencer par une lettre.
    - Le nombre maximal de caractères dépend du langage utilisé.
    - Ne pas utiliser les **mots clés**

DÉCLARATION DES VARIABLES

- **Il s'agit de créer la boîte et de lui coller une étiquette.**
- Le **nom** de la variable (l'étiquette de la boîte) obéit à des impératifs changeant selon les langages:
  - Un nom de variable correct commence impérativement par une lettre.
  - comporte des lettres et des chiffres, mais qui exclut la plupart des signes de ponctuation, en particulier les espaces.

DÉCLARATION DES VARIABLES

- Le **type** de la boite précise ce que l'on voudra mettre dedans, car de cela dépend de la **taille** de la boîte
  - Types numériques
  - Type alphanumérique
  - Type booléen

TYPES NUMÉRIQUE

Une variable destinée à recevoir des nombres.

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	$-3,40 \times 10^{38}$ à $-1,40 \times 10^{45}$ pour les valeurs négatives $1,40 \times 10^{-45}$ à $3,40 \times 10^{38}$ pour les valeurs positives
Réel double	$1,79 \times 10^{308}$ à $-4,94 \times 10^{-324}$ pour les valeurs négatives $4,94 \times 10^{-324}$ à $1,79 \times 10^{308}$ pour les valeurs positives

- Définir le type en fonction des besoins

PSEUDO-CODE

- **Variable** g **: Numérique**

ou encore

- **Variables** PrixHT, TauxTVA, PrixTTC **: Numérique**

TYPE ALPHANUMÉRIQUE

- également appelé **type caractère**, **type chaîne** ou en anglais, le **type string**
  - lettres, signes de ponctuation, espaces, ou même de chiffres.
- Le nombre maximal de caractères pouvant être stockés dans une seule variable **string** dépend du langage utilisé.

TYPE ALPHANUMÉRIQUE

- une chaîne de caractères est toujours notée entre guillemets
  - éviter la confusion:
    - entre des nombres et des suites de caractères chiffres.
    - entre le nom d'une variable et son contenu

TYPE BOOLÉEN

- Uniquement les valeurs logiques VRAI et FAUX: (TRUE et FALSE) ou des nombres (0 et 1)

- Le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit.

L'INSTRUCTION D'AFFECTATION

- affecter une variable c'est lui attribuer une valeur i.e mettre un contenu dans la boite
- En pseudo-code, l'instruction d'affectation se note avec le signe ←
  - Toto ← 24 //*Attribue la valeur 24 à la variable Toto*

- Comptabilité entre le contenant et le contenu
- Une variable désigné doit être au préalable déclarée

Exemple d'affectation

- Tutu ← Toto
- Tutu ← Toto + 4
- Tutu ← Tutu + 1

- ***Début***
  Riri ← "Loulou"
  Fifi ← "Riri"
  Fin
- ***Début***
  Riri ← "Loulou"
  Fifi ← Riri
  Fin

Exemple d'affectation

- **Variable** A **en Numérique**
  Début
  A ← 34
  A ← 12
  Fin

- **Variable** A **en Numérique**
  Début
  A ← 12
  A ← 34
  Fin

L'ordre dans lequel les instructions sont écrites va jouer un rôle essentiel dans le résultat final.

EXERCICES

- **Exercice 1.1**
- Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
- **Variables** A, B **en Entier**
  Début
  A ← 1
  B ← A + 3
  A ← 3
  Fin

Fin

CORRIGÉ

- Après La valeur des variables est :

A ← 1	A = 1	B = ?
B ← A + 3	A = 1	B = 4
A ← 3	**A = 3**	**B = 4**

EXERCICES

- **Exercice 1.2**
- Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?
- **Variables** A, B, C **en Entier**
  Début
  A ← 5
  B ← 3
  C ← A + B
  A ← 2
  C ← B - A
  Fin

CORRIGÉ

- Après — La valeur des variables est :

A ← 5	A = 5	B = ?	C = ?
B ← 3	A = 5	B = 3	C = ?
C ← A + B	A = 5	B = 3	C = 8
A ← 2	A = 2	B = 3	C = 8
C ← B – A	**A = 2**	**B = 3**	**C = 1**

EXERCICES

- **Exercice 1.3**
- Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
- **Variables** A, B **en Entier**
  Début
  A ← 5
  B ← A + 4
  A ← A + 1
  B ← A - 4
  Fin

CORRIGÉ

- Après La valeur des variables est :

A ← 5	A = 5	B = ?
B ← A + 4	A = 5	B = 9
A ← A + 1	A = 6	B = 9
B ← A – 4	**A = 6**	**B = 2**

EXERCICES

- **Exercice 1.4**
- Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?
- **Variables** A, B, C **en Entier**
  Début
  A ← 3
  B ← 10
  C ← A + B
  B ← A + B
  A ← C
  Fin

CORRIGÉ

Après	La valeur des variables est :		
A ← 3	A = 3	B = ?	C = ?
B ← 10	A = 3	B = 10	C = ?
C ← A + B	A = 3	B = 10	C = 13
B ← A + B	A = 3	B = 13	C = 13
A ← C	**A = 13**	**B = 13**	**C = 13**

Exercices

- **Exercice 1.5**
- Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?
- **Variables** A, B **en Entier**
  Début
  A ← 5
  B ← 2
  A ← B
  B ← A
  Fin
- Moralité : les deux dernières instructions permettent-elles d'échanger les deux valeurs de B et A ? Si l'on inverse les deux dernières instructions, cela change-t-il quelque chose ?

CORRIGÉ

- Après La valeur des variables est :

Après		
A ← 5	A = 5	B = ?
B ← 2	A = 5	B = 2
A ← B	A = 2	B = 2
B ← A	**A = 2**	**B = 2**

- Les deux dernières instructions ne permettent donc pas d'échanger les deux valeurs de B et A, puisque l'une des deux valeurs (celle de A) est ici écrasée.
  Si l'on inverse les deux dernières instructions, cela ne changera rien du tout, hormis le fait que cette fois c'est la valeur de B qui sera écrasée.

EXERCICES

- **Exercice 1.6**
- Plus difficile, mais c'est un classique absolu, qu'il faut absolument maîtriser : écrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.
- corrigé -

CORRIGE

- **Début**
  …
  C ← A
  A ← B
  B ← C
  Fin
- On est obligé de passer par une variable dite temporaire (la variable C).

EXERCICES

- **Exercice 1.7**
- Une variante du précédent : on dispose de trois variables A, B et C. Ecrivez un algorithme transférant à B la valeur de A, à C la valeur de B et à A la valeur de C (toujours quels que soient les contenus préalables de ces variables).
- corrigé -

CORRIGE

- **Début**
  …
  D ← C
  C ← B
  B ← A
  A ← D
  Fin
- En fait, quel que soit le nombre de variables, une seule variable temporaire suffit…

EXPRESSIONS ET OPÉRATEURS

- Une expression est un ensemble d'opérandes, reliées par des opérateurs, et équivalent à une seule valeur
- Exemple d'expression (opérandes numériques):

  7; 5+4; 123-45+844;Toto-12+5-Riri;

- Opérandes: numérique, alphanumérique, booléen…

Expressions et opérateurs

- Opérateurs numériques
  - + : addition
  - - : soustraction
  - * : multiplication
  - / : division
  - ^ : puissance
  - …
- Priorité des opérateurs: * et / sur + et -
  - *12 * 3 + 5 ≠ 12*(3+5)*

EXPRESSIONS ET OPÉRATEURS

- Opérateur alphanumérique : &
  - Variables A, B, C en Caractère
    Début
    A ← « toto"
    B ← « titi"
    C ← A & B
    Fin
- Opérateurs logiques (ou booléens)
  - Ou, ET, NON, …

Opérations logiques

- OU

A	B	A ou B
0	0	0
0	1	1
1	0	1
1	1	1

- ET

A	B	A ET B
0	0	0
0	1	0
1	0	0
1	1	1

- NON

A		NON A
0		1
1		0

EXERCICE

- **Exercice 1.8**
- Que produit l'algorithme suivant ?
- **Variables** A, B, C **en Caractères**
  Début
  A ← "423"
  B ← "12"
  C ← A + B
  Fin
- corrigé -

- Il ne peut produire qu’une erreur d’exécution, puisqu’on ne peut pas additionner des caractères.

EXERCICE

- **Exercice 1.9**
- Que produit l'algorithme suivant ?
- **Variables** A, B **en Caractères**
  Début
  A ← "423"
  B ← "12"
  C ← A & B

EXERCICE

- A la fin de l’algorithme "42312".

Test

- **Si Condition** Alors Instruction(s) **Fin si**
- **SiCondition** Alors Instruction(s)-1 **Sinon** Instruction(s)-2 **Fin si**
- **Condition = valeur logique d'une expression**

TEST IMBRIQUÉS

- **Variable** Temp **: Entier**

  Début

  Ecrire "Entrez la température de l'eau : " Lire Temp

  Si Temp <= 0 Alors

  Ecrire "C'est de la glace" FinSi

  Si Temp > 0 Et Temp < 100 Alors Ecrire "C'est du liquide"
  Finsi

  Si Temp > 100 Alors

  Ecrire "C'est de la vapeur" Finsi
  Fin

ET SI C'ETAIT DE LA GLACE ???

Et s'il y avait 723 autres tests à évaluer ???

TEST IMBRIQUÉS - NÉCESSITÉ

```
Variable Temp : 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
```

Boucles

- Tant que (Tester puis exécuter)

  Tant que condition Faire

  …

  Instructions

  FinTantQue

Boucles

- Répéter … jusqu'à (Exécuter puis tester)

  Répéter

  . . .

  Instruction(s)

  . . .

  jusqu'à condition

Boucles

- Pour (Nombre d'itérations connu au préalable)

  Pour i allant de début **à** fin

  . . .

  Instruction(s)

  . . .

  FinPour

Boucles imbriquées

- Boucle dans une boucle
- Exemple: 10 itérations telles que 6 opérations / itération

```
Variables i, j: entier Pour i allant de 1 à 10
écrire("Itération : ", i)
Pour j allant de 1 à 6
écrire("Opération : ", j, "de l'itération : ",i) Finpour
Finpour
```

TABLEAU 1D

- Structure de données statique (taille fixe)
- Un ensemble de valeurs de même type, portant le même nom de variable.
- Le nombre qui sert a repérer chaque élément du tableau s'appelle un indice.
- Un tableau de taille n est une structure constituée de n emplacements consécutifs en mémoire.
- Permet la possibilité du traitement basé sur les boucles.

TABLEAU 1D

- Pour déclarer un tableau il faut préciser le nombre et le type de valeurs qu'il contiendra.
- EX: Note[35] : réels
  - Cette déclaration réserve l'emplacement de 35 éléments de type réel.
  - Chaque élément est repéré par son indice (position de l'élément dans le tableau).
- **Convention**: la première position porte le numéro 0
  - Premier élément du tableau: Note[0]
  - Dernier élément du tableau: Note[34]

TABLEAU 1D - EXEMPLE

Nom: Calcul de la moyenne des éléments d'un tableau
Variables Note[35], i, somme : entier
moyenne **: réel**
Début
pour i allant de 0 à 34 faire
ecrire("entrer une note :")
lire(Note[i])
fin pour
somme →0 /* effectuer la moyenne des notes */
pour i allant de 0 à 34 faire
somme → somme + Note[i]
finPour
moyenne → somme / 35
ecrire("la moyenne des notes est : ",moyenne) /* afficher la moyenne */
Fin

TABLEAU 1D

- Applications:
  - Ecrire un algorithme qui cherche une valeur dans un tableau et qui retourne 1 si elle y existe et 0 sinon
  - Et si le tableau était trié ?
  - Ecrire un algorithme qui évalue un tableau et retourne 1 si ce tableau est palindrome et 0 sinon.

TABLEAU 1D DYNAMIQUE

- Examinons l'exemple suivant:

```
Variable vect[30] i, n:entier
Début
Ecrire("donner la taille du tableau (<= 30):")
lire(n)
écrire("donner les éléments du vecteur : ")
pour i allant de 0 à n-1 faire
        lire(vect[i])
Finpour
 Fin
```

et si on veut que n > 30 ???

Par mesure de **précaution, remplacer** vect[30] par vect[100]

et si on veut que n > 100 ???

TABLEAU 1D DYNAMIQUE

- Solution: **Allocation dynamique** de l'espace mémoire.
- Ce n'est qu'au **cours du programme**, qu'on va déterminer la taille de mémoire à allouer.
- **Exemple: (1/2)**

```
Variables       Notes[], moyenne, somme :réels
        i, n :entiers
Début
écrire(" Entrer le nombre de notes à saisir :")
lire(n)
/*allouer n nombres de types réels  */
allouer(Notes,n)
/* saisir les notes */  écrire("entrer    les ",n," notes :")
```

TABLEAU 1D DYNAMIQUE

- **Exemple: (2/2)**

```
pour i allant de 0 à n-1 faire
lire(Note[i])
 Finpour
/*  effectuer la moyenne des notes*/
somme      → 0
 Pour i allant  de  0    à    n-1  faire
  somme →   somme  +    Note[i]
 FinPour
  moyenne      → somme    /     n
/*  affichage de  la   moyenne     */
écrire("la moyenne des ",n," notes     est  :",moyenne)
/*  liberer     le   tableau  Notes    */
 libèrer(Notes)
Fin
```

Tableau 2D

- Pour ranger les notes de **nb** élèves de la classe d :

  matière 1 → tab_1 (nb colonnes)

  matière 2 → tab_2 (nb colonnes)

  …

  matière m → tab_m (nb colonnes)

  Si m est grand alors manipulation délicate de plusieurs variables.

- **Solution**: rassembler tout dans un seul tableau 2D de taille m lignes et nb colonnes (m x nb)

Tableau 2D

- Structure de données statique. (taille fixe)
- Un ensemble de valeurs de même type portant le même nom de variable.
- Pour repérer un élément du tableau (matrice) il **suffit** d'indiquer 2 positions: indice ligne et indice colonne.
- Permet la possibilité du traitement basé sur les boucles.

TABLEAU 2D

- Déclaration:

  Variable nom_tab[nb_ln][nb_col] : type

- Initialisation: Pour initialiser un tableau 2D (matrice) on peut utiliser les instructions suivantes :
  - T1[3][3] → {{1, 2, 3}, {4 ,5 , 6 },{ 7, 8, 9 }}
  - T2[3][3]→ {1, 2, 3, 4, 5, 6, 7, 8, 9}
  - T3[4][4]→ {{ 1, 2, 3},{ 4, 5, 6},{ 7, 8, 9}}
  - T4[4][4] → {1, 2, 3, 4 , 5, 6, 7, 8, 9}

TABLEAU 2D

- **Exemples**: Lecture, écriture et somme de matrices
  - Lecture d'une matrice:

```
Variable Tableau mat[10][10],i,j,m,n : entiers
  Début
  écrire("donner  le     nombre de ligne de la matrice       :")
  lire(m)
  écrire("donner  le nombre de colonne de la matrice :")  lire(n)
  pour      i allant      de 0  à m-1faire
      écrire("donner les éléments de la   ",i,"   ligne:")      pour j allant       de
       0      à      n-1    faire
          lire(mat[i][j])
    finpour
  Finpour
  Fin
```

TABLEAU 2D

- **Exemples**: Lecture, écriture et somme de matrices
- Ecriture d'une matrice:

```
variable Tableau Mat[10][10],i,j,n,k      :      entier
début
pour      i      allant  de    0 à n-1 faire
pour  j      allant de     0 à k-1 faire
                écrire(Mat[i][j],"")
      finpour
       écrire("\n")  /*      retour      à      la      ligne  */
finpour
fin
```

TABLEAU 2D

- **Exemples**: Lecture, écriture et somme de matrices
- Somme de deux matrices:(1/3)

```
constante N      20
Variable A[N] [N], B[N][N], O[N][N]  i,j,n : entiers
Début
Ecrire("donner la taille des matrices (<20):")
lire(n)
/*      lecture      de     la matrice A*/
pour iallant       de     0       à n-1
Ecrire("donner les éléments de la  ",i,"ligne:")
     Pour j allant de 0 à n-1        faire
  lire(A[i][j])
    Fin pour
Fin pour
```

TABLEAU 2D

- **Exemples**: Lecture, écriture et somme de matrices
- Somme de deux matrices (2/3)

```
/*  lecture  de  la  matrice B */
pour i  allant  de  0  à  n-1  faire
    écrire("donner les éléments de la ",i," ligne:")
    pour  j  allant de  0  à  n-1 faire
        lire(B [i][j])
    finpour
fipour
/*  la  somme  de  C  =  A  +  B  */
pour i  allant  de  0  à  n-1  faire
pour  j allant  de  0  à  n-1  faire
                        C  [i][j]→A  [i][j]+B [i][j]  finpour
Fipour
```

TABLEAU 2D

- **Exemples**: Lecture, écriture et somme de matrices
- Somme de deux matrices : (3/3)

```
/* affichage de la matrice de C */
    pour  i  allant    de   0     à     n-1 faire

    pour  j allant     de   0     à n-1 faire                    écrire(C[i][j],"
     ")
        finpour
            écrire("\n")     /* retour à la ligne*/
  finpour
Fin
```

Exercice: Lecture, écriture et produit de deux matrices

POINTEUR

- Un pointeur = Adresse

- Déclaration:
  - variable nom_point : ^ type
- Remarque:
- Maison de superficie 60 m^2
- Maison de superficie 300 m^2

- Taille d'un pointeur Fixe quelque soit le type de la variable pointée.

POINTEUR

- Dessin: p pointe sur a

Variable a, b: entier

Variable p: ^entier

P →&a // le pointeur p contient l'adresse de a

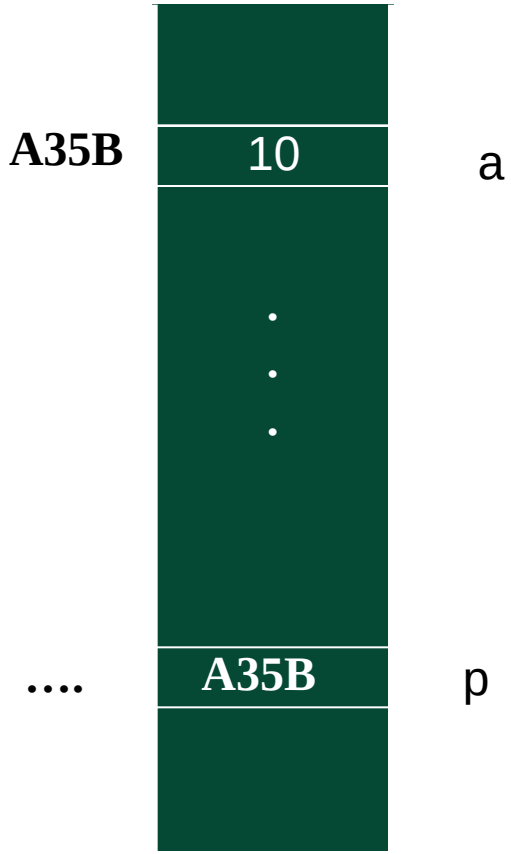
Pour accéder au contenu de la variable a via p on utilise la notation p^

Exemple :

b→p^ // équivalent à b →a, b vaut 10

POINTEUR

- La RAM



POINTEURS ET TABLEAUX 1D

- Toute opération effectuée par indexation dans un tableau est réalisable avec des pointeurs.
- **Affectation**:
  - soient **tab** et **pt** respectivement un tableau d'Entier et pointeur vers type Entier,
  - pt →tab ou pt →&tab[0] /* pt pointe sur le 1er élément de tab */
- **Remarque:** le **nom d'un tableau** (sans indices à sa suite) est un considéré comme **pointeur constant** sur le début du tableau.
- **Différences entre un tableau et un pointeur**:
  - un tableau alloue de la place en mémoire (pointeur ne le fait pas)
  - un tableau n'est pas une ***lvalue*** (alors que le pointeur l'est)

Comprendre les lvalues et les rvalues en C et C++

- Les termes lvalue et rvalue ne sont pas quelque chose que l'on rencontre souvent dans la programmation C/C++, mais quand on le fait, il n'est généralement pas immédiatement clair ce qu'ils signifient. L'endroit le plus courant pour rencontrer ces termes est dans les messages d'erreur et d'avertissement du compilateur. Par exemple, compiler ce qui suit avec gcc :

```
int foo() {return
2;}
int main() {
 foo() = 2;
return 0; }
```

- On obtient:
  - test.c: In function 'main':
  - test.c:8:5: error: lvalue required as left operand of assignment

- Certes, ce code est quelque peu pervers et n'est pas quelque chose que vous écririez, mais le message d'erreur mentionne lvalue, qui n'est pas un terme que l'on trouve généralement dans les didacticiels C/C++. Un autre exemple est la compilation de ce code avec g++

```
int& foo() { return 2; }
```

- Maintenant l'erreur est:

```
testcpp.cpp: In function 'int& foo()':

testcpp.cpp:5:12: error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'
```

- Ici encore, l'erreur mentionne une mystérieuse rvalue. Alors, que signifient lvalue et rvalue en C et C++ ?

- Cette section présente une définition intentionnellement simplifiée des lvalues et rvalues. Le reste de l'article développera cette définition. Une lvalue (valeur de localisation) représente un objet qui occupe un emplacement identifiable dans la mémoire (c'est-à-dire qui a une adresse). Les rvalues sont définies par exclusion, en disant que chaque expression est soit une lvalue, soit une rvalue. Par conséquent, à partir de la définition ci-dessus de lvalue, une rvalue est une expression qui ne représente pas un objet occupant un emplacement identifiable dans la mémoire.

Exemples

- Les termes tels que définis ci-dessus peuvent sembler vagues, c'est pourquoi il est important de voir quelques exemples simples tout de suite. Supposons que nous ayons une variable entière définie et affectée à :
  - Int var; var = 4;
- Une affectation attend une lvalue comme opérande gauche, et var est une lvalue, car il s'agit d'un objet avec un emplacement mémoire identifiable. En revanche, les éléments suivants ne sont pas valides :
- 4 = var; //ERROR
- (var + 4) = 5; //ERROR

- Ni la constante 5, ni l'expression var + 4 ne sont des lvalues (ce qui en fait des rvalues). Ce ne sont pas des lvalues car les deux sont des résultats temporaires d'expressions, qui n'ont pas d'emplacement mémoire identifiable (c'est-à-dire qu'elles peuvent simplement résider dans un registre temporaire pendant la durée du calcul). Par conséquent, leur assigner n'a aucun sens sémantique - il n'y a nulle part où assigner. Il devrait donc maintenant être clair ce que signifie le message d'erreur dans le premier extrait de code. foo renvoie une valeur temporaire qui est une rvalue. Tenter de l'affecter est une erreur, donc en voyant foo() = 2; le compilateur se plaint qu'il s'attendait à voir une lvalue sur le côté gauche de l'instruction d'affectation. Cependant, toutes les affectations aux résultats des appels de fonction ne sont pas invalides. Par exemple, les références C++ rendent cela possible :

```
int globalvar = 20 ;
int& foo() {
    return globalvar; }
int main() {
        foo() = 10;
        return 0 ; }
```

- Ici, foo renvoie une référence, qui est une lvalue, à laquelle elle peut donc être affectée.
- En fait, la capacité de C++ à renvoyer des lvalues à partir de fonctions est importante pour implémenter certains opérateurs surchargés. Un exemple courant est la surcharge de l'opérateur crochets [] dans les classes qui implémentent une sorte d'accès de recherche. std::map fait ceci :

```
std::map<int, float>

mymap; mymap[10] = 5.6;
```

L'affectation mymap[10] fonctionne car la surcharge non const de std::map::operator[] renvoie une référence à laquelle peut être affectée.

Modifiables Lvalues

- Initialement, lorsque les lvalues ont été définies pour C, cela signifiait littéralement "valeurs appropriées pour le côté gauche de l'affectation". Plus tard, cependant, lorsque ISO C a ajouté le mot-clé const, cette définition a dû être affinée. Après tout:

```
const int a = 10;    // 'a' is an lvalue

a = 10;              // but it can't be assigned!
```

- Il fallait donc ajouter un raffinement supplémentaire. Toutes les lvalues ne peuvent pas être affectées. Ceux qui peuvent être appelés lvalues modifiables. Formellement, la norme C99 définit les lvalues modifiables comme :

- [...] lvalue qui n'a pas de type tableau, n'a pas de type incomplet, n'a pas de type qualifié const, et s'il s'agit d'une structure ou d'une union, n'a aucun membre (y compris, récursivement, aucun membre ou élément de tous les agrégats ou unions contenus) avec un type qualifié const.

Conversions entre lvalues et rvalues

- D'une manière générale, les constructions de langage opérant sur des valeurs d'objet nécessitent des rvalues comme arguments. Par exemple, l'opérateur d'addition binaire '+' prend deux rvalues comme arguments et renvoie une rvalue :
- `int a = 1; // a is an lvalue`
- `int b = 2; // b is an lvalue`
- `int c = a + b; // + needs rvalues, so a and b are converted to rvalues // and an rvalue is returned`
-

- Comme nous l'avons vu précédemment, a et b sont tous les deux des lvalues. Par conséquent, à la troisième ligne, ils subissent une conversion implicite lvalue en rvalue. Toutes les lvalues qui ne sont pas des tableaux, des fonctions ou des types incomplets peuvent être converties ainsi en rvalues. Et dans l'autre sens ? Les rvalues peuvent-elles être converties en lvalues ? Bien sûr que non! Cela violerait la nature même d'une lvalue selon sa définition. Cela ne signifie pas que les lvalues ne peuvent pas être produites à partir de rvalues par des moyens plus explicites. Par exemple, l'opérateur unaire '*' (déréférencement) prend un argument rvalue mais produit une lvalue comme résultat. Considérez ce code valide :

```
int arr[] = {1, 2};
int* p = &arr[0];
*(p + 1) = 10; // OK: p + 1 is an rvalue, but *(p + 1) is an lvalue
```

Inversement, l'opérateur d'adresse unaire '&' prend un argument lvalue et produit une rvalue :

```
int var = 10;
int* bad_addr = &(var + 1); // ERROR: lvalue required as unary '&' operand
int* addr = &var;                  // OK: var is an lvalue
&var = 40;      // ERROR: lvalue required as left operand // of assignment
```

- L'esperluette joue un autre rôle en C++ - elle permet de définir des types de référence. Celles-ci sont appelées "références lvalue". Les références lvalue non constantes ne peuvent pas se voir attribuer des rvalues, car cela nécessiterait une conversion rvalue-lvalue non valide :

```
std::string& sref = std::string(); // ERROR: invalid initialization of
                                   // non-const reference of type
                                   // 'std::string&' from an rvalue of
                                   // type 'std::string'
```

- Des références lvalue constantes peuvent être affectées à des rvalues. Comme ils sont constants, la valeur ne peut pas être modifiée via la référence et il n'y a donc aucun problème à modifier une rvalue. Cela rend possible l'idiome C++ très courant d'accepter des valeurs par des références constantes dans des fonctions, ce qui évite la copie et la construction inutiles d'objets temporaires.

Arithmétique des Pointeurs

- **Règle d'ajout d'un entier** : Si on additionne un entier à un pointeur pointant sur un tableau, alors le résultat est aussi un pointeur.
- **Exemple:** si **p** est un pointeur vers **tab** un tableau d'Entier, alors:
  - p → tab; // p pointe sur le 1er élément de tab
  - p+1 pointe sur le 2ème élément de tab
  - valeur de p: adresse de tab[0]
  - valeur de p+1: adresse de tab[0] + taille(Entier)
  - valeur de p + i: adresse de tab[0] + i*taille(Entier)
- **Question:** et si p pointe vers un tableau **T** de type long ?
- **Réponse:** valeur de p + i: adresse de T[0] + i*taille(long)
- **Correspondance**:
  - pt + i correspond à &tab[i]
  - (pt + i)^ ou pt[i] correspond à tab[i]
- **Formule Générale:** **p + i correspond à p + i*taille(p^)**

ARITHMÉTIQUE DES POINTEURS

- **Soustraction de pointeurs de même type**
- Si p et q deux pointeurs sur le même type alors
- **q - p** fournit le ***nombre d'éléments du type pointé*** situés entre les deux adresses correspondantes.
- Exemple :

Variable tab[5]: Entier

p,q : ^Entier

p → tab

q → &tab[3]

Alors

q-p vaut 3

ARITHMÉTIQUE DES POINTEURS

- Comparaison de pointeurs de même type
- Soient p et q deux pointeurs sur un même tableau tab,
- La comparaison p < q est vraie si p pointe sur un élément qui précède celui sur lequel pointe q
- Ex:

```
variable   tab[10]      :      Numérique
           p,q    :      ^      Numérique
 Début
tab →     {1,2,3,4,5,6,7}
    p → tab
      q →        &tab[5]
     Tant que    (p<q) faire
           Ecrire ("\n p[",p – tab  "]:",p^)
           p → p+1
    FinTantQue  Fin
```

Arithmétique des Pointeurs

- L'exemple précédant affiche:

```
p[0]    :    1
p[1]    :    2
p[2]    :    3
p[3]    :    4
p[4]    :    5
```

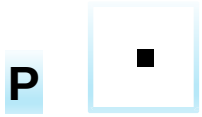
ARITHMÉTIQUE DES POINTEURS

- **Exemple:** affichage de tableau par p sans utiliser indice

```
variable tab[NB] :Réel
 p : ^Réel
Début
   p    →    tab
  pour i allant de 0 à NB-1
Ecrire("\n donner tab[",i,"]: ")
lire p^
p → p+1
 fin pour
 Ecrire ("\n les éléments du tableau:")
 p →tab
TantQue  (p < tab +    NB)    faire
   Ecrire("\ntab[",p-tab ,"]: ",p^)
FinTantQue
Fin
```

Arithmétique des Pointeurs

- **Pointeur null :** pointeur qui ne pointe sur aucune variable
- **p → NULL**
- **ou p → 0**



- **Comparaison:**
  - si (p <> NULL) ou si (p <> 0)

POINTEURS - ALLOCATION MÉMOIRE

- Allocation dynamique de la mémoire :
  - Réalisée dans le TAS (Heap en anglais) ou mémoire dynamique
  - Réalisée lors de l'éxecution du programme
  - Réservée à un pointeur p via la fonction
    - allocation(p) /* cas d'un élément */
    - allocation(p,nb_elem) /* cas de **nb_elem** éléments */
- Libération de la mémoire
  - via la fonction libère(p)
  - Libère et restitue au Système d'Exploitation la zone mémoire du tas pointée par le pointeur p

POINTEURS - ALLOCATION MÉMOIRE

- **Attention:**
  - La fonction **libère()** doit être **appliquée uniquement** à des pointeurs pointant sur une zone allouée dynamiquement (par la fonction **allocation**).
  - La fonction libère() **ne libère pas** l'espace mémoire occupé par la variable pointeur.
  - La fonction **libère()** rend indéterminée la valeur de p (p pointe vers n'importe quoi).
- **Recommandation:**
  - Après l'utilisation de la fonction libère à un pointeur, il vaut mieux le rendre **NULL**.
  - Ex:

    libère(p)

    P → NULL

Algorithme de Tri

- Les tableaux permettent de stocker plusieurs éléments de même type au sein d'une seule entité,
- Lorsque le type de ces éléments possède un ordre total, on peut donc les ranger en ordre croissant ou décroissant
- Trier un tableau c'est donc ranger les éléments d'un tableau en ordre croissant ou décroissant
- Il existe plusieurs méthodes de tri qui se différencient par leur complexité d'exécution

TRI D'UN TABLEAU : LE TRI PAR SÉLECTION

- Combien de fois au cours d'une carrière (brillante) de développeur a-t-on besoin de ranger des valeurs dans un ordre donné ?. Aussi, plutôt qu'avoir à réinventer à chaque fois la roue, vaut-il mieux avoir assimilé une ou deux techniques solidement éprouvées, même si elles paraissent un peu ardues au départ.
- Admettons que le but de la manœuvre soit de trier un tableau de 12 éléments dans l'ordre croissant. La technique du tri par sélection est la suivante : on met en bonne position l'élément numéro 1, c'est-à-dire le plus petit. Puis en met en bonne position l'élément suivant. Et ainsi de suite jusqu'au dernier. Par exemple, si l'on part de :

45	122	12	3	21	78	64	53	89	28	84	46

La procédure échanger

Tous les algorithmes de tri utilisent une procédure qui permet d'échanger (de permuter) la valeur de deux variables. Dans le cas où les variables sont entières, la procédure échanger est la suivante :

```
Procédure échanger(a,b : Entier)
        Déclaration  temp : Entier
début
    temp←a
    a←b
    b←temp
fin
```

TRI PAR SÉLECTION

- On commence par rechercher, parmi les 12 valeurs, quel est le plus petit élément , et où il se trouve. On l'identifie en quatrième position (c'est le nombre 3), et on l'échange alors avec le premier élément (le nombre 45). Le tableau devient ainsi :

3	122	12	**45**	21	78	64	53	89	28	84	46

- On recommence à chercher le plus petit élément, mais cette fois, seulement à partir du deuxième (puisque le premier est maintenant correct, on n'y touche plus). On le trouve en troisième position (c'est le nombre 12). On échange donc le deuxième avec le troisième :

3	**12**	**122**	45	21	78	64	53	89	28	84	46

- On recommence à chercher le plus petit élément à partir du troisième (puisque les deux premiers sont maintenant bien placés), et on le place correctement, en l'échangeant, ce qui donnera :

3	12	**21**	45	**122**	78	64	53	89	28	84	46

TRI PAR SÉLECTION

```
Procedure TriSelection(Tableau t[1..n]):
    variable pp, posit : entier
    n := nombre d'éléments du tableau
    Pour i variant de 0 à n-1 faire
      pp := t[i]
      posit := i
      Pour j variant de i+1 à n faire      // (t[i+1], t[i+2], ... , t[n], )
         Si t[j] < pp faire
            pp := t[j]                     // t[j] est le nouveau minimum partiel
            posit := j                     // indice ou bien position mémorisé
          fin si
      fin pour
      echange := t[posit]
      t[posit] := t[i]
      t[i] := echange                      //on échange les positions de t[i] et de pp
   fin pour
Fin procedure
```

TRI PAR SÉLECTION

- Et cetera, et cetera, jusqu'à l'avant dernier.
- En algorithmique, nous pourrions décrire le processus de la manière suivante :
  - Boucle principale : prenons comme point de départ le premier élément, puis le second, etc, jusqu'à l'avant dernier.
  - Boucle secondaire : à partir de ce point de départ mouvant, recherchons jusqu'à la fin du tableau quel et le plus petit élément. Une fois que nous l'avons trouvé, nous l'échangeons avec le point de départ.

TRI PAR SÉLECTION

Pour notre exemple:

```
// boucle principale : le point de départ se décale à chaque tour
Pour i ← 0 à 10
// on considère provisoirement que t[i] est le plus petit élément
  posmini ← i
// on examine tous les éléments suivants
    Pour j ← i + 1 à 11
          Si t[j] < t[posmini] Alors
             posmini ← j
         Finsi
         j suivant

    Fin pour
//A cet endroit, on sait maintenant où est le plus petit élément. Il ne
//reste plus qu'à effectuer la permutation.
    temp ← t[posmini]
    t[posmini] ← t[i]
    t[i] ← temp
//On a placé correctement l'élément numéro i, on passe à présent au
//suivant.
    i suivant
Fin pour
```

LA RECHERCHE DANS UN TABLEAU

- Soit un tableau comportant 20 valeurs. L'on doit écrire un algorithme saisissant un nombre au clavier, et qui informe l'utilisateur de la présence ou de l'absence de la valeur saisie dans le tableau.
- La première étape, évidente, consiste à écrire les instructions de lecture / écriture, et la boucle – car il y en a manifestement une – de parcours du tableau :

```
Tableau Tab [19] : Numérique
Variable N : Numérique
Début
    Ecrire ("Entrez la valeur à rechercher")
    Lire (N)
    Pour i ← 0 à 19
        ???
        i suivant

    Fin pour
Fin
```

LA RECHERCHE DANS UN TABLEAU

- Il nous reste à combler les points d'interrogation de la boucle Pour. Évidemment, il va falloir comparer N à chaque élément du tableau : si les deux valeurs sont égales, alors N fait partie du tableau. Cela va se traduire, bien entendu, par un Si … Alors … Sinon. Et voilà un algorithme qui fait ce travail :

```
Tableau Tab [19] en Numérique

Variable N en Numérique
Début
Ecrire ("Entrez la valeur à rechercher" )
Lire N
Pour i ← 0 à 19
  Si N = Tab[i] Alors
    Ecrire ( N, "fait partie du tableau "  )
  Sinon
    Ecrire( N, "ne fait pas partie du tableau "  )
  FinSi
i suivant
fin pour
Fin
```

LA RECHERCHE DANS UN TABLEAU

- Cet algorithme n’est pas adapté à notre problème et il est couteux:
- Il suffit d'ailleurs de le faire tourner mentalement pour s'en rendre compte. De deux choses l'une : ou bien la valeur N figure dans le tableau, ou bien elle n'y figure pas. Mais dans tous les cas, l'algorithme ne doit produire qu'une seule réponse, quel que soit le nombre d'éléments que compte le tableau. Or, l'algorithme ci-dessus envoie à l'écran autant de messages qu'il y a de valeurs dans le tableau, en l'occurrence pas moins de 20 !
- Il y a donc une erreur manifeste de conception : l'écriture du message ne peut se trouver à l'intérieur de la boucle : elle doit figurer à l'extérieur. On sait si la valeur était dans le tableau ou non uniquement lorsque le balayage du tableau est entièrement accompli.
- Nous réécrivons donc cet algorithme en plaçant le test après la boucle. Faute de mieux, on se contentera de faire dépendre pour le moment la réponse d'une variable booléenne que nous appellerons Trouvé.

LA RECHERCHE DANS UN TABLEAU

```
Tableau Tab [0..19] Numérique
Variable N Numérique
Début
Ecrire ("Entrez la valeur à rechercher")
Lire N
Pour i ← 0 à 19
    ???
i suivant
Fin pour
Si Trouve Alors
  Ecrire( "N fait partie du tableau")
Sinon
  Ecrire ("N ne fait pas partie du tableau")
FinSi
Fin
```

- Il ne nous reste plus qu'à gérer la variable Trouvé. Ceci se fait en deux étapes.
  1. un test figurant dans la boucle, indiquant lorsque la variable Trouvé doit devenir vraie (à savoir, lorsque la valeur N est rencontrée dans le tableau). Et attention : le test est asymétrique. Il ne comporte pas de "sinon".
  2. last, but not least, l'affectation par défaut de la variable Trouvé, dont la valeur de départ doit être évidemment Faux.
- Au total, l'algorithme complet – et juste ! – donne :

LA RECHERCHE DANS UN TABLEAU

```
Tableau Tab [0..19] en Numérique
Variable N en Numérique
Début
Ecrire ("Entrez la valeur à rechercher")
Lire (N)
Trouve ← Faux
Pour i ← 0 à 19
  Si (Tab[i] = N) Alors
    Trouve ← Vrai
  FinSi
i suivant
Fin pour
Si Trouve Alors
  Ecrire (N," fait partie du tableau")
Sinon
  Ecrire (N," ne fait pas partie du tableau")
FinSi
Fin
```

- Méditons un peu sur cette affaire.
- La difficulté est de comprendre que dans une recherche, le problème ne se formule pas de la même manière selon qu'on le prend par un bout ou par un autre. On peut résumer l'affaire ainsi : il suffit que N soit égal à une seule valeur de Tab pour qu'elle fasse partie du tableau. En revanche, il faut qu'elle soit différente de toutes les valeurs de Tab pour qu'elle n'en fasse pas partie.
- Voilà la raison qui nous oblige à passer par une variable booléenne , un « drapeau » qui peut se lever, mais jamais se rabaisser. Et cette technique de flag (que nous pourrions élégamment surnommer « gestion asymétrique de variable booléenne ») doit être mise en œuvre chaque fois que l'on se trouve devant pareille situation.
- Autrement dit, connaître ce type de raisonnement est indispensable et savoir le reproduire à bon escient ne l'est pas moins.
- Dans ce cas précis, il est vrai, on pourrait à juste titre faire remarquer que l'utilisation de la technique du flag, si elle permet une subtile mais ferme progression pédagogique, ne donne néanmoins pas un résultat optimum. En effet, dans l'hypothèse où la machine trouve la valeur recherchée quelque part au milieu du tableau, notre algorithme continue – assez bêtement, il faut bien le dire – la recherche jusqu'au bout du tableau, alors qu'on pourrait s'arrêter net.
- Le meilleur algorithme possible, même s'il n'utilise pas de flag, consiste donc à remplacer la boucle **Pour** par une boucle **TantQue** : en effet, là, on ne sait plus combien de tours de boucle il va falloir faire (puisqu'on risque de s'arrêter avant la fin du tableau). Pour savoir quelle condition suit le TantQue, raisonnons à l'envers : on s'arrêtera quand on aura trouvé la valeur cherchée... ou qu'on sera arrivés à la fin du tableau. Appliquons la transformation de Morgan : Il faut donc poursuivre la recherche tant qu'on n'a pas trouvé la valeur et qu'on n'est pas parvenu à la fin du tableau. Démonstration :

LA RECHERCHE DANS UN TABLEAU

```
Tableau Tab [0..19] en Numérique
Variable N en Numérique
Début
Ecrire ("Entrez la valeur à rechercher")
Lire (N)
i ← 0
TantQue (T[i] <>N et i <= 19)
   i ← i + 1
FinTantQue
Si (Tab[i] = N) Alors
   Ecrire ("N fait partie du tableau")
Sinon
   Ecrire ("N ne fait pas partie du tableau" )
FinSi
Fin
```

Exercice

- Ecrire un algorithme qui demande à l'utilisateur de saisir 30 nombres et les stocker dans un tableau (tab)
- Trier ce tableau
- Vérifier si le tableau contient la valeur 3
- Vérifier si le tableau contient 4 ou 5

TRI DE TABLEAU + FLAG = TRI À BULLES

- tri de tableau + flag = tri à bulles.
- L'idée de départ du tri à bulles consiste à se dire qu'un tableau trié en ordre croissant, c'est un tableau dans lequel tout élément est plus petit que celui qui le suit. En effet, prenons chaque élément d'un tableau, et comparons-le avec l'élément qui le suit. Si l'ordre n'est pas bon, on permute ces deux éléments. Et on recommence jusqu'à ce que l'on n'ait plus aucune permutation à effectuer. Les éléments les plus grands « remontent » ainsi peu à peu vers les dernières places, ce qui explique la charmante dénomination de « tri à bulle ». Comme quoi l'algorithmique n'exclut pas un minimum syndical de sens poétique.

TRI DE TABLEAU + FLAG = TRI À BULLES

- En quoi le tri à bulles implique-t-il l'utilisation d'un flag ?
- par ce qu'on ne sait jamais par avance combien de remontées de bulles on doit effectuer. En fait, tout ce qu'on peut dire, c'est qu'on devra effectuer le tri jusqu'à ce qu'il n'y ait plus d'éléments qui soient mal classés. Ceci est typiquement un cas de question « asymétrique » : il suffit que deux éléments soient mal classés pour qu'un tableau ne soit pas trié. En revanche, il faut que tous les éléments soient bien rangés pour que le tableau soit trié.
- Nous baptiserons le flag Yapermute, car cette variable booléenne va nous indiquer si nous venons ou non de procéder à une permutation au cours du dernier balayage du tableau (dans le cas contraire, c'est signe que le tableau est trié, et donc qu'on peut arrêter la machine à bulles). La boucle principale sera alors :

Variable Yapermute **en Booléen**
Début
…
TantQue Yapermute
…
FinTantQue
Fin

TRI DE TABLEAU + FLAG = TRI À BULLES

- Que va-t-on faire à l'intérieur de la boucle ? Prendre les éléments du tableau, du premier jusqu'à l'avant-dernier, et procéder à un échange si nécessaire. C'est parti :

```
Variable Yapermute en Booléen
Début
…
TantQue Yapermute
Pour i ← 0 à 10
  Si t[i] > t[i+1] Alors
    temp ← t[i]
    t[i] ← t[i+1]
    t[i+1] ← temp
  Finsi
Fin pour
i suivant
Fin
```

TRI DE TABLEAU + FLAG = TRI À BULLES

- Mais il ne faut pas oublier un détail capital : la gestion de notre flag. L'idée, c'est que cette variable va nous signaler le fait qu'il y a eu au moins une permutation effectuée. Il faut donc :
  - lui attribuer la valeur Vrai dès qu'une seule permutation a été faite (il suffit qu'il y en ait eu une seule pour qu'on doive tout recommencer encore une fois).
  - la remettre à Faux à chaque tour de la boucle principale (quand on recommence un nouveau tour général de bulles, il n'y a pas encore eu d'éléments échangés),
  - dernier point, il ne faut pas oublier de lancer la boucle principale, et pour cela de donner la valeur Vrai au flag au tout départ de l'algorithme.
- La solution complète donne donc :

TRI DE TABLEAU + FLAG = TRI À BULLES

```
Variable Yapermute en Booléen
Début
...
Yapermute ← Vrai
TantQue Yapermute
  Yapermute ← Faux
  Pour i ← 0 à 10
    Si t[i] > t[i+1] alors
      temp ← t[i]
      t[i] ← t[i+1]
      t[i+1] ← temp
      Yapermute ← Vrai
    Finsi

   Fin pour
 FinTantQue
Fin
```

Exercices

Ecrire un algorithme qui permet :

- De déclarer un tableau de taille 20 et le remplir
- Chercher la valeur 45 dans le tableau en utilisant deux méthodes différentes
- Trier le tableau en ordre croissant en utilisant l'algorithme de tri par sélection
- Inserer la valeur45 dans la case d'avant fin
- Trier le tableau en ordre décroissant en utilisant l'algorithme de tri à bulle

Exercices

Ecrire un algorithme qui affiche

******

*****

****

***

**

*

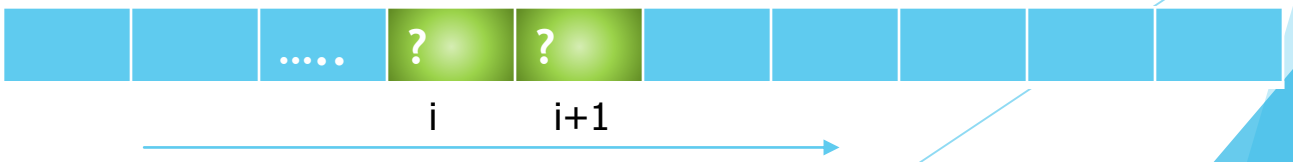
Tri à bulle

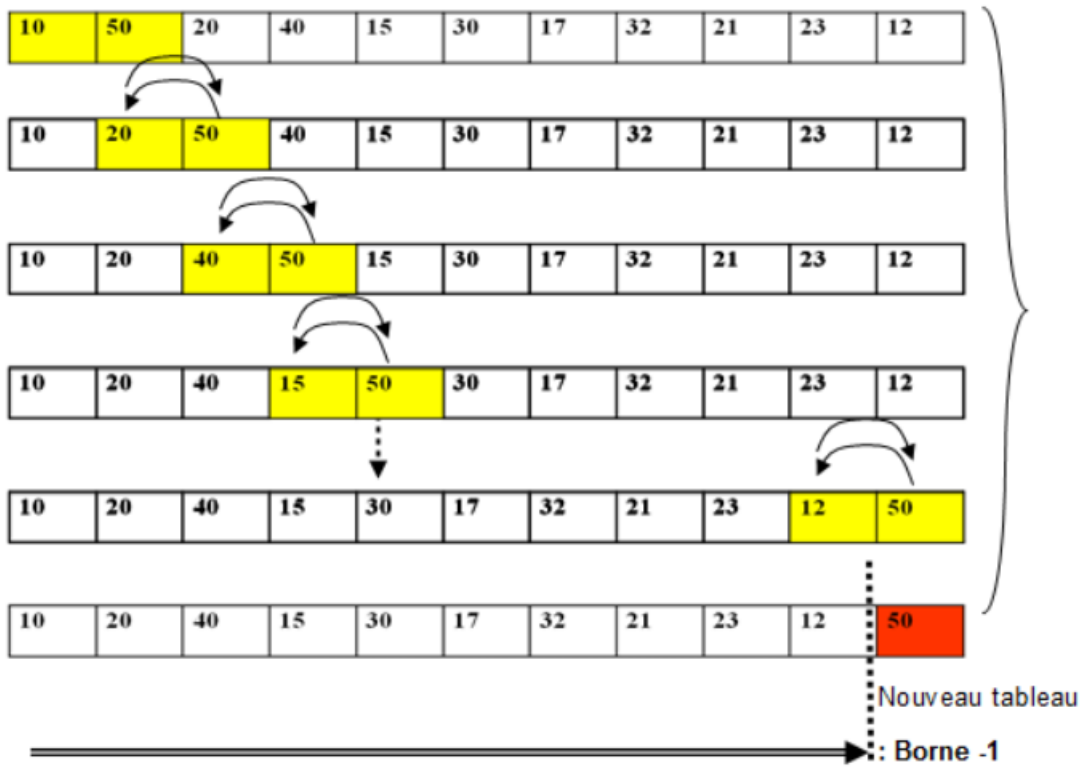
- Le **tri à bulle** consiste à parcourir le tableau, par exemple de gauche à droite, en ***comparant les éléments côte à côte*** et en les permutant s'ils ne sont pas dans le bon ordre. Au cours d'une passe du tableau, les plus grands éléments remontent de proche en proche vers la droite comme des **bulles vers la surface.**
- ***On s'arrête dès que l'on détecte que le tableau est trié : si aucune permutation n'a été faite au cours d'une passe.***

Tant que le tableau n'est pas trié

1. Effectuer une passe

1. Parcourir le tableau
2. Si 2 tableaux successifs ne sont pas dans le bon ordre, les échanger




10 50 20 40 15 30 17 32 21 23 12
10 20 50 40 15 30 17 32 21 23 12
10 20 40 50 15 30 17 32 21 23 12
10 20 40 15 50 30 17 32 21 23 12
10 20 40 15 30 17 32 21 23 12 50
10 20 40 15 30 17 32 21 23 12 50
Une passe
Nouveau tableau
: Borne -1

Algorithme tri à bulle

```
CONST TAILLE=50
TYPE TABLE=TABLEAU[TAILLE] REEL
VAR T   :TABL
        BORNE: ENTIER
        TRI: BOOLEAN
        DEBUT
                TRI→FAUX
                BORNE →TAILLE
                TANT QUE(TRI=FAUX)  FAIRE
                        TRI=VRAI
                        POUR i  de A BORNE -1  FAIRE
                                SI T[i ]>T[i+1]  ALORS
                                   Echanger(T,i,i+1)
                                   TRI→ FAUX
                                FIN SI
                        FIN POUR
                        BORNE →BORNE-1
                FIN TANT QUE
FIN
```

Cout du tri à bulle

- Le tri à bulles est l'un des tris les plus lents, si ce n'est le plus lent. C'est pour cette raison que son utilisation se fait très rare et cet algorithme reste très critiqué. Mais pourquoi est-il si lent ? On évalue la rapidité d'un algorithme de tri en observant son nombre de comparaisons/échanges et on établi ainsi une échelle que l'on nomme la complexité. Le tri à bulles fait énormément de comparaisons/échanges pour peu de valeurs à trier. On estime mathématiquement qu'il fait en moyenne n(n-1)/4 opérations pour n valeurs à trier.
- Avec la notation de Landau, on néglige le -1 et le /4 pour conserver l'opération n^2 qui croît beaucoup plus vite. C'est une croissance quadratique. En ajoutant quelques valeurs supplémentaires à trier, le rapidité de l'algorithme peut donc terriblement chuter. La complexité moyenne du tri à bulles est donc en $O(n^2)$ ce qui est extrêmement lent par rapport aux algorithmes de tri en O(n*log2(n)) tel le tri fusion.

Tri par insertion

- C'est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite...
- Le principe du tri par insertion est donc d'insérer à la $n_{ième}$ itération le $n_{ième}$ élément à la bonne place.
- L'animation ci-après illustre le fonctionnement de ce tri

Tri par insertion

```
void tri_insertion(int* t)
{
    int i, j;
    int en_cours;
    for (i = 1; i < 20; i++) {
        en_cours = t[i];
        for (j = i; j > 0 && t[j - 1] > en_cours; j--) {
            t[j] = t[j - 1];
        }
        t[j] = en_cours;
    }
}
```

LA RECHERCHE DICHOTOMIQUE

- C’est une technique célèbre de recherche, qui révèle toute son utilité lorsque le nombre d'éléments est très élevé. Par exemple, imaginons que nous ayons un programme qui doive vérifier si un mot existe dans le dictionnaire. Nous pouvons supposer que le dictionnaire a été préalablement entré dans un tableau (à raison d'un mot par emplacement). Ceci peut nous mener à, disons à la louche, 40 000 mots.
- Une première manière de vérifier si un mot se trouve dans le dictionnaire consiste à examiner successivement tous les mots du dictionnaire, du premier au dernier, et à les comparer avec le mot à vérifier. Ca marche, mais cela risque d'être long : si le mot ne se trouve pas dans le dictionnaire, le programme ne le saura qu'après 40 000 tours de boucle ! Et même si le mot figure dans le dictionnaire, la réponse exigera tout de même en moyenne 20 000 tours de boucle. C'est beaucoup, même pour un ordinateur.

LA RECHERCHE DICHOTOMIQUE

- Or, il y a une autre manière de chercher, bien plus intelligente pourrait-on dire, et qui met à profit le fait que dans un dictionnaire, les mots sont triés par ordre alphabétique. D'ailleurs, une personne qui cherche un mot dans le dictionnaire ne lit jamais tous les mots, du premier au dernier : il utilise lui aussi le fait que les mots sont triés.
- Pour une machine, quelle est la manière la plus rationnelle de chercher dans un dictionnaire ? C'est de comparer le mot à vérifier avec le mot qui se trouve au milieu du dictionnaire. Si le mot à vérifier est antérieur dans l'ordre alphabétique, on sait qu'on devra le chercher dorénavant dans la première moitié du dictionnaire. Sinon, on sait maintenant qu'on devra le chercher dans la deuxième moitié.
- A partir de là, on prend la moitié de dictionnaire qui nous reste, et on recommence : on compare le mot à chercher avec celui qui se trouve au milieu du morceau de dictionnaire restant. On écarte la mauvaise moitié, et on recommence, et ainsi de suite.

LA RECHERCHE DICHOTOMIQUE

- A force de couper notre dictionnaire en deux, puis encore en deux, etc. on va finir par se retrouver avec des morceaux qui ne contiennent plus qu'un seul mot. Et si on n'est pas tombé sur le bon mot à un moment ou à un autre, c'est que le mot à vérifier ne fait pas partie du dictionnaire.
- Regardons ce que cela donne en terme de nombre d'opérations à effectuer, en choisissant le pire cas : celui où le mot est absent du dictionnaire.
  - Au départ, on cherche le mot parmi 40 000.
  - Après l'itération n°1, on ne le cherche plus que parmi 20 000.
  - Après l'itération n°2, on ne le cherche plus que parmi 10 000.
  - Après l'itération n°3, on ne le cherche plus que parmi 5 000.
  - etc.
  - Après l'itération n°15, on ne le cherche plus que parmi 2.
  - Après l'itération n°16, on ne le cherche plus que parmi 1.

LA RECHERCHE DICHOTOMIQUE

- Et là, on sait que le mot n'existe pas. On a obtenu notre réponse en 16 opérations contre 40 000 précédemment !. Attention, la recherche dichotomique ne peut s'effectuer que sur des éléments préalablement triés.

exemple

- Soit le tableau suivant:

12	13	15	17	20	23	28	29	33	40

  - chercher la valeur ***23*** si elle existe donner son index

RECHERCHE DICHOTOMIQUE

```
algorithme CHERCHER-DICHO(T,x):entier;

données T[1,n]: tableau trié; x: valeur;
variables g,d,m: entiers;
debut
   g ← 1;
   d ← n;
   m ← (g + d)/2;
   Tant Que ((g < d) ET (x ≠ T[m])) FAIRE
            SI (x < T[m]) ALORS
               d ← m - 1;
            SINON g ← m + 1;
            Fin Si
            m ← (g + d)/2;
    Fin Tant Que
    SI (T[m] = x) ALORS
        retourner m;
    SINON
    retourner 0;
    FSI
fin
```

Les Fonctions Prédéfinies

Tout langage de programmation propose un certain nombre de **fonctions** ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs et pénibles algorithmes.

STRUCTURE GÉNÉRALE DES FONCTIONS

- Exemple: A ← Sin(35)
- Une fonction est donc constituée de trois parties :
- le **nom** proprement dit de la fonction. Ce nom ne s'invente pas ! Il doit impérativement correspondre à une fonction proposée par le langage. Dans notre exemple, ce nom est SIN.
- deux parenthèses, une ouvrante, une fermante. Ces parenthèses sont toujours **obligatoires**, même lorsqu'on n'écrit rien à l'intérieur.
- une liste de valeurs, indispensables à la bonne exécution de la fonction. Ces valeurs s'appellent des **arguments**, ou des **paramètres**. A noter que même dans le cas de ces fonctions n'exigeant aucun argument, les parenthèses restent obligatoires. Le nombre d'arguments nécessaire pour une fonction donnée ne s'invente pas : il est fixé par le langage. Par exemple, la fonction sinus a besoin d'un argument (ce n'est pas surprenant, cet argument est la valeur de l'angle). Si vous essayez de l'exécuter en lui donnant deux arguments, ou aucun, cela déclenchera une erreur à l'exécution. Notez également que les arguments doivent être d'un certain **type**, et qu'il faut respecter ces types.

LES FONCTIONS DE TEXTE

- Tous les langages, proposent les fonctions suivantes, même si le nom et la syntaxe peuvent varier d'un langage à l'autre :

Len(chaîne) : renvoie le nombre de caractères d'une chaîne

Mid(chaîne,n1,n2) : renvoie un extrait de la chaîne, commençant au caractère n1 et faisant n2 caractères de long.

LES FONCTIONS DE TEXTE

- Ce sont les deux seules fonctions de chaînes réellement indispensables. Cependant, pour nous épargner des algorithmes fastidieux, les langages proposent également :

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.

LES FONCTIONS DE TEXTE

- Exemples :
- Len("Bonjour, ça va ?") vaut 16
  Len("") vaut 0
  Mid(" langage C est facile", 4, 7) vaut "gage C "
  Mid("C++ est l'incrémentation de C", 12, 1) vaut "n"
  Left("Et pourtant…", 8) vaut "Et pourt"
  Right("Et pourtant…", 4) vaut "t…"
  Trouve("Un pur bonheur", "pur") vaut 4
  Trouve("Un pur bonheur", "techno") vaut 0
- Il existe aussi dans tous les langages une fonction qui renvoie le caractère correspondant à un code Asci donné (fonction **Asc**),

  Asc("N") vaut 78

TROIS FONCTIONS NUMÉRIQUES CLASSIQUES

- **Partie Entière**
- Une fonction extrêmement répandue est celle qui permet de récupérer la partie entière d'un nombre :
- Après : A ← Ent(3,228) A vaut 3
- **Modulo**
- Cette fonction permet de récupérer le reste de la division d'un nombre par un deuxième nombre. Par exemple :
- A ← Mod(10,3) A vaut 1 car 10 = 3*3 + 1
  B ← Mod(12,2) B vaut 0 car 12 = 6*2
  C ← Mod(44,8) C vaut 4 car 44 = 5*8 + 4
  - **Génération de nombres aléatoires**

  Après : Toto ← Alea() On a : 0 <=Toto < 1

Trois fonctions numériques classiques

- Par exemple, si je veux générer un nombre entre 1,35 et 1,65 ; la « fourchette » mesure 0,30 de large. Donc : 0 =< Alea()*0,30 < 0,30

Il suffit dès lors d'ajouter 1,35 pour obtenir la fourchette voulue. Si j'écris que :

Toto ← Alea()*0,30 + 1,35

Les fonctions de conversion

- Dernière grande catégorie de fonctions, là aussi disponibles dans tous les langages, car leur rôle est parfois incontournable, les fonctions dites de conversion.

  fonctions de cast …

Les Fichiers

- 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.
- Imaginons que l'on veuille écrire un programme gérant un carnet d'adresses. D'une exécution du programme à l'autre, l'utilisateur doit pouvoir retrouver son carnet à jour, avec les modifications qu'il y a apportées la dernière fois qu'il a exécuté le programme. Les données du carnet d'adresse ne peuvent donc être inclues dans l'algorithme, et encore moins être entrées au clavier à chaque nouvelle exécution !

Les Fichiers

- Les **fichiers** sont là pour combler ce manque. **Ils servent à stocker des informations de manière permanente, entre deux exécutions d'un programme.** Car si les variables, 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…).

Les Fichiers

- Un fichier est un ensemble de données. Il peut servir soit à la lecture, pour rentrer des informations dans un programme , soit à l'écriture pour sauvegarder les résultats obtenus.
- Les fichiers sont caractérisés par deux notions :
  - le mode d'organisation : comment sont organisées les données dans le fichier (séquentiel, indexé,...) ;
  - le mode d'accès : comment sont accédées les données dans le fichier (séquentiel, direct, binaire,...).

Les Fichiers

- Ces caractéristiques sont étroitement liées aux langages de programmation utilisés. Chacun de ces derniers offre différents types de fichiers.
- En algorithmique, nous nous limiterons par souci de facilité aux fichiers texte et aux fichiers d'enregistrements.
- L'utilisation d'un fichier se fait selon les phases suivantes :

  Ouverture du fichier.

  Traitement du fichier.

  Fermeture du fichier.

LES FICHIERS TEXTE

- Les fichiers de type Texte sont des fichiers séquentiels (mode d'organisation séquentielle). Les informations sont disposées de façon séquentielle, les unes à la suite des autres. Elles ne sont ni en ligne ni en colonne ! Elles sont repérées par un pointeur. Leur organisation est séquentielle et leur accès ne peut être que séquentiel .

- Trois opérations sont définies sur ce type de fichiers :

La lecture : Lors de l'ouverture du fichier, le pointeur *pointe* sur la 1ière information, quelque soit son type. A chaque accès (***Lire***), le pointeur se déplace sur l'information suivante, (mode d'accès séquentiel). Si on veut lire une information en amont du pointeur, il faut fermer le fichier, le rouvrir et lire jusqu'à l'information désirée.

L'écriture : un fichier non vide ouvert en écriture perd tout ce qu'il possède. En effet, dès son ouverture le pointeur est positionné sur la première ligne. Seules les opérations d'écriture sont autorisées.

L'ajout : cette opération permet de rajouter de nouvelles données à la fin du fichier sans détruire ce qu'il y avait auparavant. Le pointeur est positionné sur la marque de fin de fichier qui est décalée d'une position après chaque rajout

- *Exemple :*

Var Fich1 : FICHIER texte

Pour i := 1 à 100 faire

Lire (Fich1, Tab[i])

fin Pour

Les données du fichier *Fich1* sont lues et stockées dans le tableau *Tab*.

Ecrire (Fich1, ‘’Le salaire annuel est de : “, *Sal_Annuel) ;*

Ces informations sont stockées dans *Fich1* à l’endroit où se trouve le pointeur lors de cette action.

Ajout (Fich1, “Le salaire annuel est de : “, *Sal_Annuel) ;*

Ces informations sont rajoutées à la fin de *Fich1* et le pointeur reste sur la marque de fin de fichier.

- Il n'y a que la fin du fichier qui est marquée par un symbole repéré par la fonction *EOF*(*Nomfichier*), qui rend la valeur *vraie* si elle le rencontre, la valeur *faux* sinon.
- Un fichier séquentiel ne peut être ouvert qu'on lecture ou en écriture. Après l'ouverture d'un fichier, la première opération (***Lire***, ***Ecrire*** ou ***Ajout***) indique si le fichier est accessible en lecture ou en écriture.
- ***Ecrire*** et ***Ajout*** sont des opérations d'écriture, leur seule différence et due au fait que pour ***Ecrire***, le pointeur se place en début du fichier, alors que pour ***Ajout***, il se place en fin du fichier.

- Toute manipulation d'un fichier nécessite 3 phases :

" Ouverture du fichier :

OUVRIR (Nomfichier)

" Traitement du fichier : Lecture ou Ecriture :

LIRE(Nomfichier, ...) ;

ECRIRE(Nomfichier, ...) ;

AJOUT(Nomfichier, ...) ;

" Fermeture du fichier :

FERMER(Nomfichier)

- ***Remarques :***

La fonction ***EOF(Fich1)*** permet de tester si le pointeur est sur la fin du fichier *Fich1*.

L'utilité des fichiers est la sauvegarde les données.

- Il est préférable d'utiliser ***Ecrire*** à la place d'***Afficher***, quand on utilise les fichiers.

LES FICHIERS D'ENREGISTREMENTS

- C'est un ensemble d'enregistrements (ou d'articles) de type structuré que l'on déjà défini.
- *Exemple :*

```
Type Etudiant = Enregistrement
                    Numéro : entier ;
                    NomPrénom : Chaîne[30]
                    Discipline : Chaîne[25]
                    Année_Inscrip : 1950..2000
                fin ;
Var E1 : Etudiant
Fich_Etudiant : FICHIER de Etudiant ;
E1.Numéro := 134561 ;
E1.NomPrénom := ''Dupont Lionel'' ;
E1.Discipline := ''Sciences économiques'' ;
E1.Année_Inscrip := 1996
Afficher(Fich_Etudiant, E1) ;
```

- On peut traiter un fichier d'enregistrements de manière séquentielle, mais son intérêt et de permettre un accès direct aux données.
- Lors de l'ouverture d'un tel fichier, le pointeur est positionné sur le premier enregistrement. On peut se déplacer directement sur n'importe quel enregistrement avant une opération de lecture ou d'écriture à l'aide de l'action :

Positionner(Fichier, N° enregistrement)

- ***Remarques :***

Contrairement aux fichiers séquentiels, un fichier d'enregistrements peut être ouvert en lecture et en écriture.

La taille d'un fichier de ce type est le nombre de ses enregistrements.

La complexité

- "Si l’on veut résoudre un problème à l’aide d’un ordinateur, il est indispensable d’avoir recours à un algorithme car un ordinateur ne peut exécuter que des ordres précis et sans ambiguïté.". Stockmeyer et Chandra.
- Pour résoudre donc un problème avec un ordinateur, il vous faut un algorithme. Or pour pouvoir l’étudier et le comparer avec d’autres, il faut utiliser la notion de complexité.
- Nous donnerons les notions de base sur les différentes classes de problèmes et quelques conseils sur l’optimisation des programmes.
- Enfin, nous ferons une présentation de notions mathématiques sur le comportement asymptotique des fonctions qui sont utilisées lors de l’étude de la notion de complexité algorithmique.

Définition

- La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.
- Pour Stockmeyer et Chandra, "l'efficacité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction du nombre des données."
  - Nous avons donc deux éléments à prendre en compte :
    - la taille des données ;
    - le temps de calcul.

- Note importante : contrairement à ce que le nom suggère, la complexité n'est pas une mesure de si un algorithme est « simple » ou « complexe » d'un point de vue humain. C'est en fait bien souvent l'inverse : un algorithme simple aura généralement une complexité plus élevée (il « prend plus de temps ») qu'un algorithme ingénieux, qui aura une faible complexité (plus « rapide »)

La taille des données

La taille des données (ou des entrées) va dépendre du codage de ces entrées.

On choisit comme taille la ou les dimensions les plus significatives.

Par exemple, en fonction du problème, les entrées et leur taille peuvent être :

- ✓ des éléments : le nombre d'éléments;
- ✓ des nombres : nombre de bits nécessaires à la représentation de ceux-là;
- ✓ des polynômes : le degré, le nombre de coefficients non nuls;
- ✓ des matrices m × n : max(m,n), m.n, m + n;
- ✓ des graphes : nombre de sommets, nombre d'arcs, produit des deux ;
- ✓ des listes, tableaux, fichiers : nombre de cases, d'éléments;
- ✓ des mots : leur longueur.

Le temps de calcul

- Le temps de calcul d'un programme dépend de plusieurs éléments :
  - ✓ la quantité de données;
  - ✓ mais aussi de leur encodage ;
  - ✓ de la qualité du code engendré par le compilateur ;
  - ✓ de la nature et la rapidité des instructions du langage ;
  - ✓ de la qualité de la programmation ;
  - ✓ et de l'efficacité de l'algorithme.

- Nous ne voulons pas mesurer le temps de calcul par rapport à toutes ces variables. Mais nous cherchons à calculer la complexité des algorithmes qui ne dépendra ni de l'ordinateur, ni du langage utilisé, ni du programmeur, ni de l'implémentation. Pour cela, nous allons nous mettre dans le cas où nous utilisons un ordinateur RAM (Random Access Machine):
  - ordinateur idéalisé;
  - mémoire infinie;
  - accès à la mémoire en temps constant;
  - généralement à processeur unique (pas d'opérations simultanées).

Pour connaître le temps de calcul, nous choisissons une opération fondamentale et nous calculons le nombre d'opérations fondamentales exécutées par l'algorithme.

Opération fondamentale

- C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.
  Par exemple :

Problème	Opération fondamentale
Recherche d'un élément dans une liste	Comparaison
Tri d'une liste, d'un fichier, ...	Comparaisons, déplacement
Multiplication des matrices réelles	Multiplications et additions
Addition des entiers binaires	Opération binaire

Coût des opérations

- Coût de base

Pour la complexité en temps, il existe plusieurs possibilités :

  - première solution : calculer (en fonction de n) le nombre d'opérations élémentaires (addition, comparaison, affectation, ...) requises par l'exécution puis le multiplier par le temps moyen de chacune d'elle ;
  - pour un algorithme avec essentiellement des calculs numériques, compter les opérations coûteuses (multiplications, racine, exponentielle, ...) ;
  - sinon compter le nombre d'appels à l'opération la plus fréquente (cf. tableau précèdent).

- Ici, nous adoptons le point de vue suivant : l'exécution de chaque ligne de pseudo code demande un temps constant. Nous noterons c_i le coût en temps de la ligne i

- Voyons un algorithme simple qui renvoie la somme de tous éléments d'un ensemble représenté dans un tableau avec longueur explicite.

```
somme(E)
    s →0
    pour i de 0 à E.longueur -1 faire
            s → s + E.tab[i]
    retourner s
```

- Analysons les différentes parties de cet algorithme :
  - L'initialisation de s coûte 1 opération ;
  - pour la boucle, le nombre d'opérations est la somme des opérations de chacune des itérations :
    - corps de boucle : 2 opérations : une addition puis le stockage du résultat dans s ;
    - maintenance de boucle : à chaque fois un incrément et un test sont cachés dans le « pour » : i →i+1 et i <= longueur -1 : 2 opérations
    - entrée dans la boucle : initialisation de i et le premier test (2 opérations, pas d'incrément au début) ;
  - retour de la fonction : 1 opération.

Coût des opérations

- Au final, chaque itération de la boucle exécute le même nombre d'instructions, le nombre total pour la boucle est donc « coût d'une itération » * « nombre d'itérations » ; la boucle est répétée « longueur » fois. La formule pour cet algorithme est 1+2+n*(2+2)+1=4+4n où n = longueur de l'ensemble.
- Nous voyons ici que le temps d'exécution de cet algorithme croît linéairement en n. Quand nous étudions la complexité algorithmique, ce qui nous intéresse le plus est le comportement asymptotique de l'algorithme, c'est-à-dire la « forme » de la formule pour n grand. Nous gardons donc seulement le « plus gros» terme de la formule sans sa constante : nous utilisons la notation « grand *O* » des mathématiques, et disons que l'algorithme est en *O*(n).

La notation grand *O*

- Étant donnés deux fonctions f et g, f = *O*(g) ssi pour tout n suffisamment grand, il existe une constante C telle que f(n)<=C*g(n). Plus formellement :

$$\exists n_0 >= 0, \exists C > 0 \text{ t.q. } \forall n >= n_0, f(n) <= C * g(n)$$

- La constante *C* de la notation nous permet d'« oublier » la constante du plus grand facteur. L'analyse devient non seulement plus simple mais surtout possible car il est en pratique presque impossible de savoir exactement combien d'opérations sont nécessaires : cela dépend notamment du processeur où le code est exécuté. Par exemple dans le corps d'une boucle, ce qui nous importe est alors juste de savoir qu'il y a un nombre constant d'opérations, et si la boucle est répétée n fois on a alors *O*(n)

- Ces deux propriétés combinées nous permettent de nous concentrer sur le comportement pour des « grandes » instances, où il y a une énorme différence par exemple entre une complexité en $O(n)$ et une en $O(n^2)$, mais où des complexités qui seraient $\approx 4*n$ et $\approx 5*n$ sont considérées comme équivalentes.
- Revenons à notre exemple pour en faire directement l'analyse avec la notation O. Pour mémoire, rappelons qu'un nombre constant d'opération est en $O(1)$. Nous avons maintenant les propriétés suivantes pour une analyse de la complexité en temps. Le détail est à gauche mais on général on annote directement sur l'algorithme comme montré dans le diapo suivant à droite

-- initialisation : $O(1)$;
-- boucle : n fois le corps de boucle ;
-- corps de boucle : $O(1)$;
-- finalisation : $O(1)$.

```
somme(E)
    s ← 0                                          O(1)
    pour i de 0 à E.longueur −1 faire              n ×
        s ← s + E.tab[i]                           O(1)
    retourner s                                    O(1)
```

Au global, la complexité est $O(1)+n*O(1)+O(1) = O(1)+O(n) = O(n)$

Comparaison les complexités

- Le but de l'analyse de complexité est de pouvoir comparer plus facilement différents algorithmes qui effectuent la même tâche. On préférera par exemple l'algorithme qui s'exécute en $O(n)$ par rapport à celui en $O(n^2)$.
- Il devient également possible de prédire le temps que prendra un algorithme à trouver la solution sur une instance très grande en extrapolant une mesure de temps d'exécution faite sur une plus petite instance.
- Dans cette section, nous nous intéressons à observer combien de fois plus de temps est nécessaire à un algorithme quand on double la taille des données d'entrée. Par exemple, un algorithme linéaire ($O(n)$) mettra deux fois plus de temps. On dénote *log n* le logarithme en base 2, qui fait partie des complexités courantes pour un algorithme (voir sections suivantes). Nous donnons ici les complexités dans l'ordre du plus courant au moins courant (sans être exhaustif, il existe bien sûr beaucoup d'autres complexités possibles). Nous donnons également une idée de la taille maximale que peuvent avoir les données (colonne « Max n ») avant que l'algorithme devienne impraticable (cela prendrait trop de temps pour résoudre le problème).

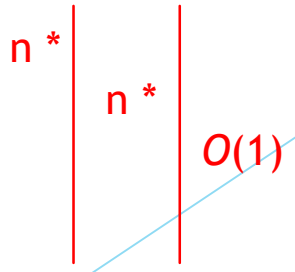
Complexité	Nom courant	Temps quand on double la taille de l'entrée	Max n
$O(n)$	linéaire	prend 2 fois plus de temps	10^{12}
$O(1)$	constant	prend le même temps	pas de limite
$O(n^2)$	Quadratique	prend 4 fois plus de temps	10^6
$O(n^3)$	cubique	prend 8 fois plus de temps	10 000
$O(\log n)$	Logarithmique	prend seulement une étape de plus	$10^{10(12)}$
$O(n \log n)$	linearithmique	prend deux fois plus de temps + un petit peu	10^{11}
$O(2^n)$	Exponentiel	prend tellement de temps que c'est inconcevable	30

On observe une séparation nette entre la complexité exponentielle et les autres, qu'on appelle polynomiales.
Les algorithmes exponentiels sont si catastrophiques qu'il ne sert à rien de se demander « que se passe-t-il si je double la taille de mon entrée » car, à l'inverse, il suffit d'augmenter la taille de 1 pour doubler le temps de calcul ! Ces algorithmes doivent être évités à tout prix, sauf si l'on sait que les instances sont très petites.

Complexité des structures de boucles et conditions

- Complexité des boucles imbriqués
  - Comme établi précédemment, pour une boucle on fait la somme des complexités de chaque itération. Dans le cas de boucles imbriquées, on calculera d'abord la complexité de la boucle interne car on en a besoin pour connaître le coût d'une itération de la boucle externe. De fait, souvent on va multiplier entre elles le nombre d'itérations de chacune des boucles
  - Considérons l'algorithme ci-dessous qui affiche toutes les additions possibles de deux nombres entre 1 et n.

```
Toutes_additions(n)
    pour i de 1 à n faire
        pour j de 1 à n faire
            afficher(i+j)
```



Complexité des structures de boucles et conditions

- On peut remarquer que cet algorithme affiche plusieurs fois chaque addition. En effet on aura par exemple 2+7 (quand i =2 et j =7) puis plus tard 7+ 2. Analysons l'algorithme ci-dessous qui ne répète pas les doublons en faisant s'arrêter j à la valeur *i* au lieu de *n*.

```
Toutes_additions(n)
        pour i de 1 à n faire                    n *
                pour j de 1 à i faire                i *
                        afficher(i+j)                        O(1)
```

Cette fois, la boucle interne a un coût de $O(i)$. Le problème est que i change à chaque itération, donc on ne peut pas simplement multiplier le tout par n : la dernière itération coûte bien $O(n)$ mais la première seulement $O(1)$!

Dans ce cas, on peut cependant être conservateur. Comme i <= n, on a i = $O(n)$ d'après la définition et on se retrouve comme dans le premier cas avec une complexité en $O(n^2)$

Complexité des structures de boucles et conditions

- Si l'on veut savoir plus précisément combien de fois la boucle interne est exécutée, il est possible de le calculer directement : à la première itération de la boucle externe, 1 fois, puis 2 fois, puis 3, etc. jusque n. Au total, cela fait $\sum_{K=1}^{N} K$ fois, soit, n*(n+1)/2. La complexité est donc n*(n+1)/2 *$O(1)$ = $O(n^2/2 + n/2)$ = $O(n^2/2) = O(n^2)$. Cette analyse est plus précise mais ne change pourtant pas la complexité qui reste en $O(n^2)$

Complexité des conditions

- Toutes les parties d'un algorithme ne sont pas forcément exécutées, c'est notamment le cas lors qu'il y a des conditions « si. . . alors. . . sinon ». L'une des deux branches sera exécutée mais pas l'autre. Il faut alors analyser séparément les deux branches possibles et les comparer. Plusieurs cas de figure sont possibles :
  - complexités égales dans le alors et le sinon : la complexité globale est alors la même. Exemple : $O(1)$ et $O(1)$ donne $O(1)$:
  - complexité plus importante dans une des deux branches : pour être conservateur, on garde alors la plus grande. Exemple $O(1)$ et $O(n)$ donne $O(n)$;
  - complexité plus importante dans une des deux branches mais celle-ci est moins souvent exécutée. On peut aussi rester conservateur et garder le maximum, mais on aura une analyse moins précise. Nous allons étudier ci-dessous un exemple.

Complexité des conditions

```
exemple (E)
    pour i de 0 à E.longueur-1 faire                n *
        si i = 0 alors
            x → E.tab[i]                                    O(1)
            pour j de 0 à E.longueur-1 faire        n *
                E.tab[j] → x+E.tab[j]                               O(1)
        sinon
            afficher (E.tab[i])                             O(1)
```

La complexité semble être en $O(n^2)$
mais c'est en réalité du $O(n)$.

Complexité des conditions

- Dans cet exemple, on pourrait en première approximation dire que la complexité est en $O(n^2)$ à cause des boucles imbriquées. Ce n'est pas faux mais cela manque de précision car ne capture pas le fait que la boucle interne n'est exécutée qu'une seule fois, lorsque i = 0. Si l'on regarde le coût des itérations de la boucle sur i, la première itération coûte bien $O(n)$, cependant les suivantes ne coûtent que $O(1)$. La complexité totale est donc de $O(n)+(n-1)*O(1) = O(n)$.

Exemples d'analyses de complexité

- Nous savons déjà que de faire la somme des éléments d'un ensemble (ou une séquence) peut être fait en temps linéaire. De manière générale, le parcours d'un ensemble ou d'une séquence (i.e., itérer sur tous les éléments) peut être effectué en O(n).
- Ajout en début de séquence
  - Cette analyse dépend de la structure de donnée utilisée pour stocker la séquence
    - liste chaînée : on ajoute une nouvelle cellule au début et on la lie à l'ancienne tête : $O(1)$;
    - tableau + longueur : on décale tous les éléments existant d'une case vers la droite : $O(n)$.

Exemples d'analyses de complexité

- Ajout en fin de séquence
  - Cela dépend encore une fois de la structure de donnée utilisée.
    - liste chaînée : on parcourt toute la liste pour arriver en queue : $O(n)$;
    - tableau + longueur : on incrémente la longueur et écrit au dernier indice : $O(1)$
- Tri par sélection
  - Dans un tri par sélection, on cherche le maximum (ou minimum) de la séquence, puis on le place à la fin (resp. au début). On répète le même processus avec le reste de la sous-séquence qui est non-triée. Il est possible de faire un tri par sélection dans un tableau ou dans une liste chaînée

Exemples d'analyses de complexité

- Dans un tableau

```
tri_selection(S)
  pour dernier de S.longueur-1 à 1 faire                    n *
        imax →0                                                  O(1)
        max → S.tab[0]                                           O(1)
        pour i de 1 à dernier faire                              O(n)*
              si S.tab[i] → max alors                                  O(1)
                    imax → i                                           O(1)
                    max → S.tab[i]                                     O(1)
        S.tab[imax] → S.tab[dernier]                             O(1)
S.tab[dernier] → max                                             O(1)
```

Exemples d'analyses de complexité

- L'algorithme utilise deux boucles imbriquées. La boucle extérieure a n itération, et celle intérieure v itération, avec v qui varie : n-1 au début mais diminue ensuite au cours du temps jusque 1. Nous savons que v <= n est toujours vrai, donc nous savons que le corps de la boucle interne est exécuté à $O(n)$ fois à chaque itération. Le corps boucle est en $O(1)$ (ainsi que le reste des calculs), donc la complexité globale est $O(n^2)$. De même que vu précédemment, on pourrait faire une analyse plus précise qui nous conduirait à trouver qu'on exécute le corps de la boucle interne n*(n-1)/2 fois, mais cela ne changerait pas la complexité

- Dans une liste chaînée

```
tri_selection(S)                                                    
    dernier ← Nil                                                   O(1)
    tant que dernier ≠ S.tête faire                                 n ×
        max ← S.tête.valeur                                             O(1)
        maxcel ← S.tête                                                 O(1)
        maxpred ← Nil                                                   O(1)
        cel ← maxcel.suivant                                            O(1)
        queue ← maxcel                                                  O(1)
        tant que cel ≠ dernier faire                                    O(n) ×
            si cel.valeur > max alors                                       O(1)
                max ← cel.valeur                                            O(1)
                maxcel ← cel                                                O(1)
                maxpred ← queue                                             O(1)
            queue ← cel                                                     O(1)
            cel ← cel.suivant                                               O(1)
        si queue ≠ maxcel alors                                         O(1)
            si maxpred = Nil alors
                S.tête ← maxcel.suivant                                 O(1)
            sinon
                maxpred.suivant ← maxcel.suivant                        O(1)
            maxcel.suivant ← dernier                                    O(1)
            queue.suivant ← maxcel                                      O(1)
```

- Analyse de complexité : chaque boucle « tant que » sur la liste chaînée agit de manière similaire à la boucle « pour » correspondante dans la représentation par tableau. Comme précédemment, on ne connait pas exactement le nombre d'itérations de la boucle interne, mais celui-ci est borné par n donc $O(n)$ itérations, et au final la complexité est $O(n^2)$.

Recherche d'un élément dans une séquence

- Considérons deux cas : selon que la séquence est triée ou non.
  - Séquence non triée Il nous faut vérifier chaque élément de la séquence : c'est un parcours classique, donc $O(n)$ pour un tableau ou une liste chaînée.
  - Séquence triée Pour une liste chaînée, cela nous permet de nous arrêter dès que l'on trouve un élément strictement plus grand que celui que l'on cherche. Cela ne change tout de même pas la complexité en général, car dans le pire des cas, il nous faut toujours vérifier jusqu'au dernier élément de la séquence. En moyenne on vérifiera la moitié de la séquence, ce qui est toujours du $O(n)$.

Recherche d'un élément dans une séquence

- Pour un tableau, nous pouvons utiliser la dichotomie puisque nous avons directement accès au « milieu » de la séquence : en comparant avec l'élément du milieu, on n'a plus ensuite qu'à chercher dans une moitié de la séquence. Supposons que l'on cherche x, et que S.tab[longueur/2] contient v ; si x < v, alors x ne peut pas être à un indice plus grand que longueur/2. En utilisant cette propriété, on peut alors faire une recherche binaire, algorithme présenté à droite.

```
recherche (S, x)                                   /* S doit être trié */
  i → 0                                                    O(1)
  j → S.longueur -1                                        O(1)
  tant que i<= j faire                                     k*
        m → (i+j) div 2                                          O(1)
        si x = S.tab[m] alors retourner m                        O(1)
        sinon si x < S.tab[m] alors j → m-1                      O(1)
        sinon i → m +1                                           O(1)
  retourner -1                                       /* non trouvé
*/
```

- Quelle est la complexité de cet algorithme ? Nous voyons que les parties avant et après la boucle sont en O(1), de même que le corps de boucle. La complexité va donc être directement liée au nombre d'itérations de cette boucle que l'on note k. Considérons la quantité j-i : au début, elle est égale à S.longueur, et à chaque itération cette quantité va être divisée approximativement par 2. Dans le pire des cas (quand x n'est pas dans la séquence), ce processus est répété jusqu'à atteindre un, puis zéro, puis i devient plus grand que j (à cause du-1 ou du +1).
- Nous devons donc répondre à la question suivante : combien de fois faut-il diviser n par 2 pour attendre 1 (on peut ne pas considérer les deux dernières étapes car on cherche une notation *O*) ? La réponse est *O*(log n) (le logarithme en base 2). En effet, on cherche le plus petit k tel que $n/2^k <= 1$, c'est-à-dire, $n <= 2^k$, et nous savons que $2^{\log n} = n$.
- Une autre façon de voir (ou prédire) la complexité logarithmique est de remarquer que même si l'on double la taille de l'entrée (i.e., une séquence de taille 2n au lieu de n), on n'a besoin que d'une étape de plus pour effectuer la recherche. . .
- En général, on retrouve une complexité logarithmique dans tous les algorithmes qui contiennent une boucle divisant une quantité de donnée par une constante à chaque itération.

Calcul des nombres de Fibonacci

- Les nombres de Fibonacci sont définis ci-dessous à gauche. Un algorithme très naïf pour calculer le $n^{ème}$ nombre est proposé à droite
- F0 =1
  F1 = 1
  Fn=Fn-1+Fn-2

```
Fibo(n)
  si n < 2 alors
      retourner 1
      sinon
        retourner Fibo(n-1) + Fibo(n-2)
```

- Il est plus difficile de calculer la complexité de cet algorithme car il est récursif : la complexité de Fibo dépend. . . de la complexité de Fibo !
- Définissons alors C_n comme étant le nombre d'opérations nécessaire pour calculer Fibo(n). On peut écrire la formule suivante : $C_n = O(1) + C_{n-1} + C_{n-2}$. Remarquez que cette formule ressemble très fortement à l'équation de récurrence de la suite de Fibonacci elle-même !
- Faisons maintenant une approximation. Puisque Fibo(n-1) va elle-même appeler Fibo(n-2), on sait que $C_{n-1} >= O(1)+C_{n-2}$, donc on écrit Cn <= 2 *Cn-1. Avec $C_0 = C_1 = 1$, on a directement que $C_n <= 2^n$, donc la complexité de Fibo est $O(2^n)$.
- Bien sûr, cette borne n'est pas exacte puisque nous avons fait une approximation grossière. La complexité réelle pourrait être bien plus petite ! Pour prouver que cet algorithme n'est pas polynomial, nous pouvons également dire que $C_n >= 2 * Cn-2$ et donc $C_n >= 2^{n/2}$. Cet algorithme a donc une complexité exponentielle, comprise entre $O(2^{n/2})$ et $O(2^n)$.

Tri fusion

- Note : l'analyse suivante est assez complexe. Elle vous permet de voir quand une complexité en O(n log n) peut apparaître. Cependant, la technique générale de l'utilisation de l'arbre des appels récursifs est importante à connaître. Nous allons voir un algorithme différent pour trier une séquence, basé sur l'idée suivante : si la séquence est de longueur supérieure à 2, on divise la séquence en deux moitié de taille identiques, on trie (récursivement) les deux sous-séquences, puis on les fusionne pour obtenir la séquence triée.
- L'implémentation présentée ici n'est pas la plus optimisée, il serait par exemple possible d'effectuer moins de copies entre tableaux. Mais cela rend difficile la lecture de l'algorithme sans pour autant en diminuer la complexité. Nous donnons une implantation à base de tableaux mais il est possible d'utiliser des listes chaînées

Tri fusion

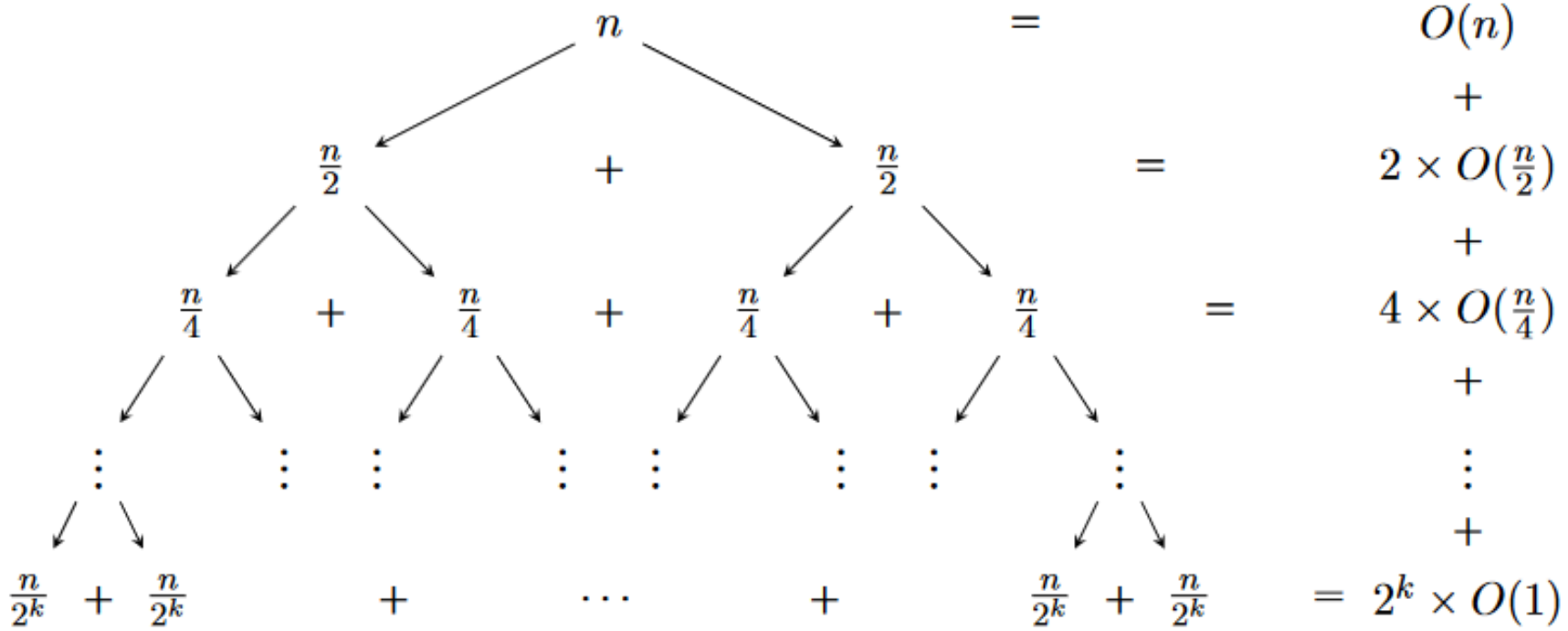
- Tri(S)
  si S.longueur = 1 alors
  retourner
  m → S.longueur div 2
  S1 →m premiers éléments de S
  S2 → les autres
  Tri(S1)
  Tri(S2)
  Fusion(S, S1, S2)

```
Fusion(S, S1, S2)
    i ← 0; i1 ← 0; i2 ← 0
tant que i1 < S1.longueur et i2 < S2.longueur faire
    si S1.tab[i1] < S2.tab[i2] alors
        S.tab[i] ← S1.tab[i1]
        i1 ← i1 + 1
    sinon
        S.tab[i] ← S2.tab[i2]
        i2 ← i2 + 1
    i ← i + 1
si i1 = S1.longueur alors
    échanger S1 et S2
    échanger i1 et i2
pour j de i1 à S1.longueur − 1 faire
    S.tab[i] ← S1.tab[j]
    i ← i + 1
S.longueur ← S1.longueur + S2.longueur
```

- L'algorithme de tri fusion utilise une stratégie classique dite « diviser pour régner ». Elle consiste en découper un gros problème en plus petits problèmes, résoudre récursivement les problèmes, puis combiner les résultats pour obtenir une solution au problème initial. Analysons la complexité de cet algorithme.

- La fonction Tri utilise la fonction Fusion, nous allons donc d'abord analyser cette dernière. Il y a deux boucles : un « tant que » et un « pour », et on ne sait pas à l'avance combien d'opération chacune d'elle va effectuer. En revanche, on sait que le nombre total d'itérations combiné des deux boucles est exactement n =S1.longueur+S2.longueur. Puisque les deux boucles font un nombre constant d'opérations, la complexité de la fonction est O(n). En effet, nous copions ici tous les éléments de S1 et S2 vers S exactement une fois. Voyons maintenant la fonction Tri, qui est récursive. Comme dans la section précédente, écrivons la formule. Si Cn est le nombre d'opérations faites par Tri sur une séquence de taille n, nous avons $C_n = O(n)+2*C_{n/2}+O(n)$.
- Le premier $O(n)$ correspond à la séparation en S1 et S2 et le deuxième leur fusion. Nous allons maintenant dessiner « l'arbre d'appels » d'une exécution de Tri, où la valeur de chaque nœud représente la taille de la séquence d'un appel (récursif) à Tri.

Tri fusion


n = $O(n)$
+
$\frac{n}{2}$ + $\frac{n}{2}$ = $2 \times O(\frac{n}{2})$
+
$\frac{n}{4}$ + $\frac{n}{4}$ + $\frac{n}{4}$ + $\frac{n}{4}$ = $4 \times O(\frac{n}{4})$
+
⋮
+
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + ⋯ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$ = $2^k \times O(1)$

- Le coût du tri est la somme des coûts de tous les nœuds de cet arbre d'appels. Le coût d'un nœud est maintenant simplement le coût de la séparation et de la fusion, puisque le coût des deux appels récursifs est maintenant compté dans les nœuds des fils.
- Si l'on regarde les coûts par niveau, on se rend compte que le premier niveau est $O(n)$, le deuxième deux fois $O(n/2)$, le troisième quatre fois $O(n/4)$, etc. Chaque niveau coûte donc $O(n)$, où n est la taille de la séquence initiale, jusqu'au dernier niveau, où il y a 2^k feuilles représentant le coût de « trier » des sous-séquences de taille 1, i.e., $O(1)$, et $2^k=n$.
- Au total, la complexité de l'algorithme du tri-fusion est $O(n)$* le nombre de niveaux, c'est-à-dire la hauteur de l'arbre. Comme pour la recherche binaire, on trouve que cette hauteur est exactement le nombre de fois qu'il faut diviser n par 2 pour atteindre 1, soit $k = \log n$. La complexité du tri fusion est donc en $O(n \log n)$, ce qui est bien meilleur que le tri par sélection analysé plus précédemment.

Exercices

- Calculer la complexité du programme suivant: complexité constante

```
Int main()
{
    Printf("veuillez entrer la valeur de n: \n");
    scanf("%d",&n);
    If(n%2 == 0)
        Printf("le nombre est pair");
    Else
        Printf("le nombre est impair");
}
```

$O(1)$

$O(1)$

$O(1)$

$O(1)$

Le maximum des deux O

Exercices

```
Int sum(int t[20])
{
        int i, s;
        s=0;
        For(i=0;i<20;i++){
            s=s+t[i];
        }
        Return s;
}
```

$O(1)$

$O(1)$

$O(1)$

$O(1)$

$(O(1)+O(1)+O(1))*20=60 \rightarrow O(60)$

→

- Ecrire ("veuillez saisir un nombre:")
- Lire(n)
- Pour i→1 à n pas 1 Faire
  - Pour j → 1 à i èas 1 Faire
    - Ecrire("Bonjour")
  - Fin pour
- Fin pour
- Fin

$O(1)$

$O(n)$

$1+2+3+..+n-1+n=n*(n-1)/2$
$\rightarrow O(n^2/2 - n/2) \rightarrow O(n^2)$ complexité quadratique

Exercices

- Pour i→1 à n pas 1 Faire
  - …
- Fin pour

$O(n)$

- Pour i→1 à n pas 2 Faire
  - …
- Fin pour

$O(n/2) \rightarrow O(n)$

- Pour i→1 à n pas i*2 Faire
  - …
- Fin pour

Si n = 10
ité1: i=1
ité2: i=2
ité3: i=4
ité4: i=8

1+log2(10)
$O(\log(n))$

Si n = 300
ité1: i=1
ité2: i=2
ité3: i=4
ité4: i=8
ité:5 i=16
ité6 i=32
uté7 i= 64
ité8 i=128
ité9 i=256

- Pour i→n à -1 pas i/2 Faire
  - …
- Fin pour

Boucle infinie → pas de complexité

- Pour i→1 à n pas pow(2,n) Faire
  - …
- Fin pour

$O(3*2^n+2)$ → $O(2^n)$ → complexité exponontiel

Exercices

- Calculer la complexité du programme suivant:

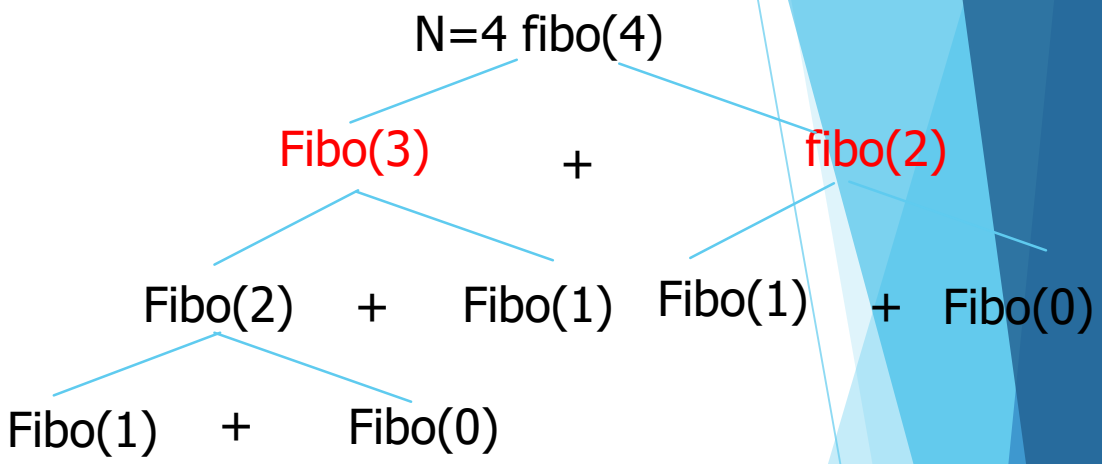
```
Long fn(n int){
        If (n <= 0)
                Return 1;
        Else
                Return 1 + fn(n/5);
}
```

N=100	N=1000
Fn(100)	Fn(1000)
Fn(20)	Fn(200)
Fn(4)	Fn(40)
3 appels	Fn(8)
	Fn(1)
	5 appels

1 + log5(100)
1 + 2,86 = 3,86

➔ $O(2+\log5(n)$ ➔ $O(\log(n))$

```
long fibo(long n)
{
    If(n<2)
        Return n;
    Else
        Return fibo(n-1)+fibo(n-2)
}
```



N=4 → 2^3=8 appels
N=5→ 2^4=16 appels

$O(2^{n-1})$ → $O(2^n)$ Complexité exponentielle
→donc cet algorithme est couteux pour les grandes entrées !!!!

```
...
Lire(n)                                          O(1)
Pour i→1 à n pas 1 Faire                         *n
    T→1                                          O(1)
    Tant que t <= n Faire                        *(1+log2(n))
        Ecrire("Bonjour")                        O(1)
        t→t*2                                    O(1)
    Fin tant que                                 O(2)
Fin pour
...
```

Si n=10
Ité 1 t→1
Ité 2 t→2
Ité 3 t→4
Ité 4 t→8

1+log2(10)

Si n=300
Ité 1 t→1
Ité 2 t→2
Ité 3 t→4
Ité 4 t→8
Ité 5 t→16
Ité 6 t→32
Ité 7 t→64
Ité 8 t→128
Ité 9 t→256

$O(n\log(n))$ complexité quasi-linéaire