Algorithmique
Module INFO 0401
2ème année informatique
Reims – Sciences Exactes

Cours n° 2

A.HEBBACHE

Complexité d'un algorithme

1. Introduction
2. La complexité des algorithmes
3. La O notation

   3. 1 Opérations

      - Somme
      - Produit
      - Les algorithmes récursifs

1.Introduction

- Un algorithme est un ensemble d'instructions permettant de transformer un ensemble de données en un ensemble de résultats et ce, en un nombre fini d'étapes.
- Pour atteindre cet objectif, un algorithme utilise deux ressources d'une machine:

  - **le temps**
  - **et l'espace mémoire**

1.Introduction

- La complexité temporelle d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

- La complexité spatiale d'un algorithme est l'espace utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

1.Introduction

➢ Facteurs affectant le temps d'exécution

1. Machine
2. Langage
3. Programmeur
4. Compilateur
5. Algorithme et structure de données.

Le temps d'exécution dépend de la taille de l'entrée.

2.La complexité d'un algorithme

2.1 Définition

La complexité d'un algorithme est un critère d'évaluation qui permet d'en anticiper ses performances.

Cette méthode permet aussi de comparer entre deux algorithmes résolvant le même problème.

2.La complexité d’un algorithme

2.1 Définition

Le temps d ’exécution d'un programme dépend des facteurs suivants:

- les données du programme
- la qualité du code généré par le compilateur
- la machine (vitesse et nature des instructions)
- complexité de l'algorithme

2.La complexité d’un algorithme

Remarque

Le fait que le temps d'exécution dépend des données signifie que le temps d’exécution pourrait être défini comme une fonction des données ou comme une fonction de n.

2.La complexité d'un algorithme

Exemple :

Recherche d'un entier dans un tableau trié

On envisage deux méthodes de recherche :

- **Recherche séquentielle depuis le début du tableau trié.**
- **Une recherche dichotomique**:

  On compare l'entier situé au milieu du tableau.

  En cas d'égalité ***c'est fini***,

  Sinon si l'entier recherché est inférieur, ***il suffit de continuer la recherche dans la première moitié du tableau***;

  s'il est supérieur, ***il suffit de rechercher dans la deuxième partie du tableau.***

On peut itérer ce procédé jusqu'à arriver à une partie du tableau réduite à un élément.

2.La complexité d'un algorithme

➢ Remarque

D'une manière intuitive on peut voir que la deuxième méthode est plus performante que la première :

- **Dans le premier cas chaque test réduit l'espace de recherche d'un élément.**
- **Dans le deuxième cas chaque test réduit de moitié.**

2.La complexité d’un algorithme

Remarque

On représente le temps d’exécution d'un programme avec n données par **`T(n)`** .

`T(n)` pourrait être le nombre d'instructions exécutées dans un ordinateur

En pratique, le temps moyen est souvent difficile à déterminer.

2.La complexité d'un algorithme

➢ Complexité au pire

C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à *n*.

- **Avantage** : il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué *T*max(*n*) opérations.
- **Inconvénient** : cette complexité peut ne pas refléter le comportement "usuel" de l'algorithme, le pire cas pouvant ne se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le pire cas.

2.La complexité d'un algorithme

➢ Complexité « en moyenne»

Dans la complexité moyenne, le point de vue adopté est probabiliste :

plutôt que de calculer une quantité en considérant la pire situation, c'est-à-dire la pire configuration des données d'entrée, on considère de toutes les configurations possibles, chacune associée à une probabilité, et on fait une somme pondérée, par ces probabilités, des valeurs prise par cette quantité dans les différentes configurations.

2.La complexité d'un algorithme

➢ Complexité au meilleur :

C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée ici à n.

C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

3.O notation

3.1 Définition

- Quand on dit que le temps d'exécution d'un algorithme est $O(n^2)$, cela veut dire :

  qu'il existe une constante $c > 0$ et une constante $n0 > 0$

  tel que pour tout $n > n0$: $T(n) \leq c\, n^2$

- La notation O exprime la borne supérieure

3.O notation

3.2 Opérations

- **Somme**

  Cette règle peut être utilisée pour calculer le temps d'exécution d'une séquence d'étapes d'un programme. Chaque étape peut être un fragment de programme avec des boucles et des branchements.

- Supposons que nous ayons trois étapes dont les temps d'exécution sont respectivement O(n2), O(n3) et O(nlogn). Alors le temps d'exécution des deux premières étapes exécutées séquentiellement est O(max(n2, n3)) = O (n3).

- Le temps d'exécution des trois étapes est

  O(max(n3, nlogn)) = O(n3).

3. O notation

3. 2 Opération

- En général, le temps d'exécution d'une séquence fixe d ’étapes et celui de l ’étape qui a le plus grand temps d'exécution.

- **Observation**

  si g(n) <= f(n) pour tout n > n0 alors
  O(f(n) + g(n)) = O (f(n))

  Ex : $O(n^2 + n) = O(n^2)$

3. O notation

3. 2 Opération

- Affectation, lecture ou écriture est O(1).

- Le temps d ’exécution d'une séquence d'instruction est déterminée par la règle de la somme. C'est donc le temps de la séquence qui a le plus grand temps d ’exécution.

- Le temps d'exécution d'une instruction IF est le temps d ’exécution des instructions exécutées sous condition, plus le temps pour évaluer la condition. Ce dernier est O(1).

- Le temps d ’exécution d'une boucle est la somme du temps pour évaluer le corps et du temps pour évaluer la condition. ce dernier prend O(1).

3. O notation

3.2 Opération

Exemple

```
bubble ( int A[n] )
int i, j, temp ;

(1) for (i=1; i<n-1; i++)
(2) {
(3)  for(j = n; j<i+1; j--)
     {
       if (A[j-1] > A[j])
       {
(4)      temp = A[j-1];
(5)      A[j-1] = A[j];
(6)      A[j] = temp;
       }
     }
   }
```

où n est le nombre de données à trier.

3. O notation

3.2 Opération

Exemple

Chaque instruction d'affectation prend une valeur constante du temps indépendante de n. Donc les instructions (4) (5) et (6) occupent chacun O(1).

C'est à dire qu'il existe $c > 0$ et $n0 > 0$ telles que :

quelque soit $n > n0$ $T(n) \leq c.1$

En d'autres termes on peut toujours trouver une constante c telle que le temps d'exécution de l'affectation soit inférieur à c

D'après la règle de la somme le temps d'exécution de (4) (5) et (6) est

$O(\max(1, 1, 1)) = O(1)$

3. O notation

3. 2 Opération

Exemple

- Considérons maintenant les instructions conditionnelles et répétitives en allant du niveau le plus interne vers le niveau le plus externe.

  Pour l'instruction IF, le test de la condition exige O(1).

  L'exécution des 3 affectations dépend de la valeur du test. puisque nous recherchons le temps d'exécution dans le cas le plus défavorable, donc l'instruction IF prend aussi O(1).

3.O notation

3. 2 Opération

Exemple

- Analysons maintenant la boucle (2) à (6).

  La règle générale pour une boucle est que le temps d'exécution est la somme du temps dépensé par l'exécution du corps de la boucle pour chaque itération. le corps de la boucle prend O(1) pour chaque itération (incrémentions de l'index, test des limites, branchement vers le début de la boucle).

  Le nombre d'itérations est n-i donc d'après la règle du produit

  corps : O(1)
  boucle : O(n-i)

  Le temps dépensé de (2) a (6) est O((n-i)*1) = O (n-i).

3. O notation

3. 2 Opération

Exemple

Analysons la boucle la plus externe qui contient toutes les instructions exécutables du programme.

L'instruction 1 est exécutée (n-1) fois.

Donc le temps total d'exécution du programme est limité par :

Somme des (n-i) pour i=1

$n-1 = n(n-1)/2 = n^2 - n/2$ qui est $O(n^2)$

Autre façon de procéder :
Dans la boucle (1) (6)

corps : O(n-i) ou O(n) (résultat précédent)
boucle : O(n-1) ou O(n)

Règle du produit : $O(nxn) = O(n^2)$

3. O notation

➢ Exemple de complexité

Tri par sélection

Voici l'algorithme de tri par sélection

```
fonction TRI_SELECTION(tableau t[n])
{
  pour i=1 à n-1 faire
  {
    min=i;
    pour j=i+1 à n faire
      si t[j] < t[min] alors
      {
        min=j;
        echanger(t[i], t[min]);
      }
  }
}
```

3. O notation

➢ **Exemple de complexité**

La boucle intérieure demande un temps $A(n - i) + B$;

cette boucle est parcourue pour $i = 1, 2, \ldots, n - 1$.

Le temps total est donc $An(n - 1)/2 + Bn = O(n^2)$.

Toutes les opérations sont à effectuer quel que soit le tableau.

Cette complexité est donc la même pour tous, et donc aussi en moyenne.

La place requise (version itérative, tri sur place) en plus du tableau est constante.

3. O notation

➢ Les tris en n log n

Le premier est une illustration du principe "diviser pour régner".

On ramène la résolution d'un problème de taille n à la résolution du même problème sur deux occurrences de taille divisée par 2.

Ceci ne réduit le temps que si la complexité du rassemblement des solutions des problèmes de taille plus petite est simple.

3. O notation

➢ Les tris en n log n

Parmi les exemples les plus significatifs, on trouve, la recherche :

- dichotomique
- l'exponentiation rapide
- le tris par tas
- et la transformée de Fourier rapide.
- etc.

3. O notation

3.3 Les algorithmes récursifs

Considérons les deux exemples suivants :

Exemple :

Fact(n) :
Si n ≤ 1 Alors Fact = 1
Sinon Fact = n * Fact(n-1) Fin-Si

Dans l'exemple 1, on peut exprimer T(n) par

$$T(n) = \begin{cases} c \text{ si } n \leq 1 \\ c + T(n-1) \text{ sinon} \end{cases}$$

3. O notation

3.3 Les algorithmes récursifs

tri d'une liste L de n éléments avec n = 2k.

```
Tri(L, n)
Si n = 1 Alors Tri = L
Sinon          L1 = L [1..n/2]
               L2 = L [(n+1)/2..n]
               Tri = Fusion( Tri(L1, n/2), Tri(L2, n/2))
Fin
```

Dans cet exemple, on peut exprimer T(n) par :

$$T(n) = \begin{cases} a & \text{si } n=1 \\ 2\,T(n/2) + bn & \text{Sinon} \end{cases}$$

C'est ce qu'on appelle les équations de récurrence.

3. O notation

3.3 Les algorithmes récursifs

Pour mesurer un algorithme récursif, il faut d'abord déterminer son équation de récurrence, puis résoudre des équations de récurrence.

Il existe en général 3 méthodes :

a) Par substitution de proche en proche.(dilater la récurrence)
b) Deviner une solution f(n) et la démontrer par récurrence.
c) Utiliser la solution de certaines équations de récurrence connues.

Les Tableaux

1. Les tableaux à une dimension

1.1 Déclaration et mémorisation

1.2. Initialisation et réservation automatique

1.3. Accès aux composantes

2. Les tableaux à deux dimensions

2.1. Déclaration et mémorisation

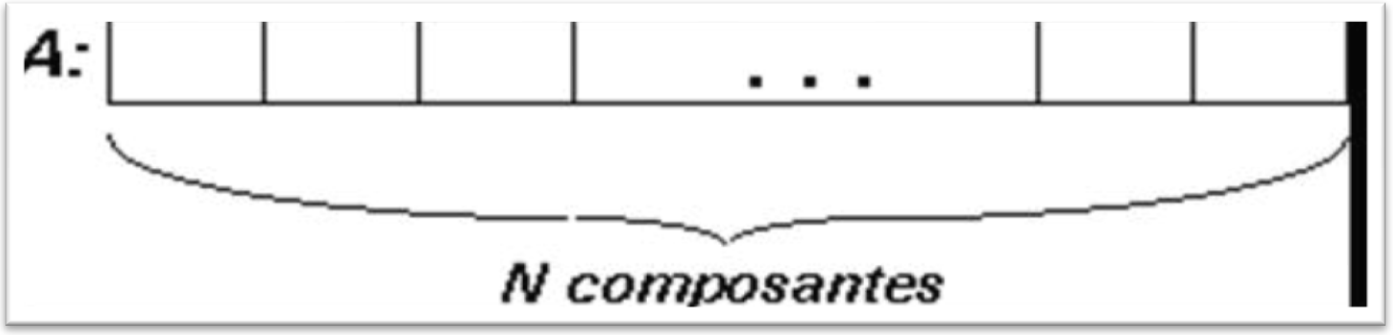
2.2. Initialisation et réservation automatique

2.3. Accès aux composantes

1. Les tableaux à une dimension

Définitions

Un tableau (uni-dimensionnel) A est une variable structurée formée d'un nombre entier N de variables simples du même type, qui sont appelées les ***composantes*** du tableau. Le nombre de composantes N est alors la ***dimension*** du tableau.



En faisant le rapprochement avec les mathématiques, on dit encore que "A est un vecteur de dimension N"

1. Les tableaux à une dimension

Exemple

La déclaration

```
int JOURS[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

définit un tableau du type **int** de dimension 12.

Les 12 composantes sont initialisées par les valeurs respectives 31, 28, 31, ... 31.

On peut accéder à la première composante du tableau par **JOURS[0]**, à la deuxième composante par **JOURS[1]**, ... à la dernière composante par **JOURS[11]**.

1. Les tableaux à une dimension

1.1 Déclaration et mémorisation

Déclaration

```
<TypeSimple> <NomTableau>[<Dimension>];
```

Les noms des tableaux sont des *identificateurs* qui doivent correspondre aux restrictions définies précédemment.

1. Les tableaux à une dimension

Exemples

Les déclarations suivantes en langage algorithmique :

tableau entier A[25]

tableau réel B[100]

tableau booléen C[10]

tableau caractère D[30]

se laissent traduire en C par:

`int A[25];`	ou	`long A[25];`	ou	...
`float B[100];`	ou	`double B[100];`	ou	...
`int C[10];`				
`char D[30];`				

1. Les tableaux à une dimension

Mémorisation

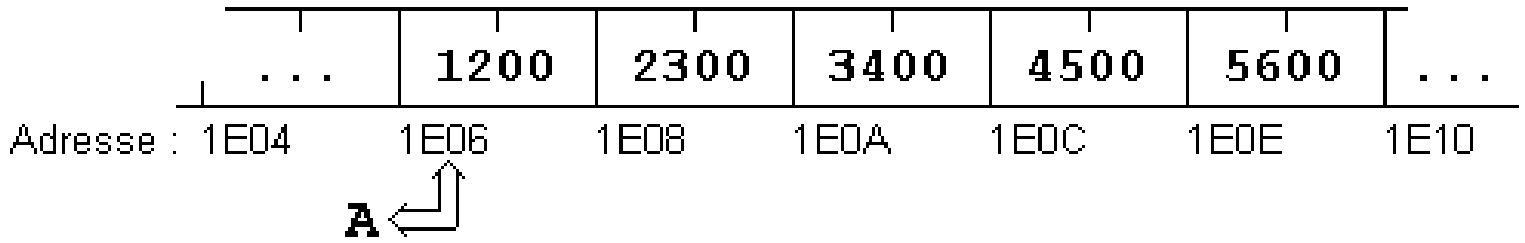
En C, le nom d'un tableau est le représentant de ***l'adresse du premier élément*** du tableau.

Les adresses des autres composantes sont calculées (automatiquement) relativement à cette adresse.

1. Les tableaux à une dimension

Exemple

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```



Si un tableau est formé de N composantes et si une composante a besoin de M octets en mémoire, alors le tableau occupera de N*M octets.

Exemple

En supposant qu'une variable du type long occupe 4 octets (c.-à-d : sizeof(long)=4), pour le tableau T déclaré par : long T[15];

C réservera N*M = 15*4 = 60 octets en mémoire.

1. Les tableaux à une dimension

1.2. Initialisation et réservation automatique

Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades (réservation prédéfinie).

Exemples

```
int A[5] = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 87e-5, -12.3E4};
int C[10]  = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

Il faut évidemment veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées par zéro.

1. Les tableaux à une dimension

Réservation automatique

Si la dimension n'est pas indiquée explicitement lors de l'initialisation, alors l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

1. Les tableaux à une dimension

Exemples

```
int A[] = {10, 20, 30, 40, 50};
```

==> réservation de **5*sizeof(int)** octets (dans notre cas: 10 octets)

```
float B[] = {-1.05, 3.33, 87e-5, -12.3E4};
```

==> réservation de 4*sizeof(float) octets (dans notre cas: 16 octets)

```
int C[] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

==> réservation de 10*sizeof(int) octets (dans notre cas: 20 octets)

```
short A[] = {1200, 2300, 3400, 4500, 5600};
```

A:	1200	2300	3400	4500	5600

```
short A[5] = {1200, 2300, 3400};
```

A:	1200	2300	3400	0	0

1. Les tableaux à une dimension

1.3. Accès aux composantes

En déclarant un tableau par:

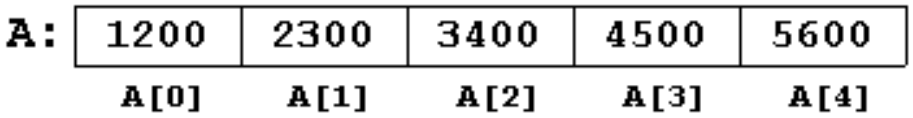
```
int A[5];
```

nous avons défini un tableau A avec cinq composantes, auxquelles on peut accéder par:

```
A[0], A[1], ... , A[4]
```

Exemples

```
short A[5] = {1200, 2300, 3400, 4500, 5600};
```




```
MAX = (A[0]>A[1]) ? A[0] : A[1];
A[4] *= 2;
```

1. Les tableaux à une dimension

Remarque

Considérons un tableau T de dimension **N** :

En C :

- l'accès au premier élément du tableau se fait par T[0]
- l'accès au dernier élément du tableau se fait par T[N-1]

En langage algorithmique :

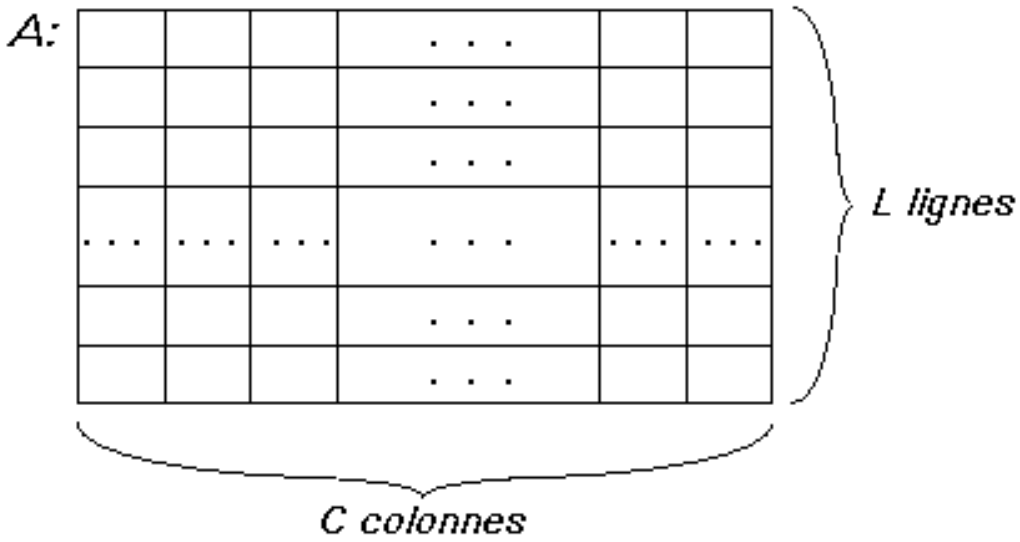
- l'accès au premier élément du tableau se fait par T[1]
- l'accès au dernier élément du tableau se fait par T[N]

2. Les tableaux à deux dimensions

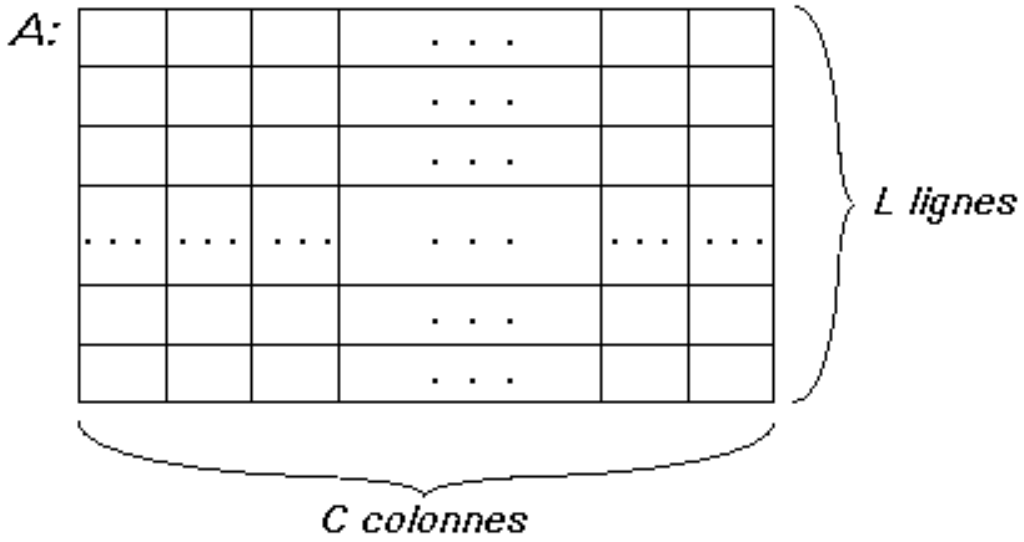
Définitions

En C, un tableau à deux dimensions A est à interpréter comme un tableau (uni-dimensionnel) de dimension L dont chaque composante est un tableau (uni-dimensionnel) de dimension C.

On appelle L le ***nombre de lignes*** du tableau et C le ***nombre de colonnes*** du tableau. L et C sont alors les deux ***dimensions*** du tableau. Un tableau à deux dimensions contient donc ***L*C composantes***.



2. Les tableaux à deux dimensions



On dit qu'un tableau à deux dimensions est ***carré***, si L est égal à C.

En faisant le rapprochement avec les mathématiques, on peut dire que *"A est un vecteur de L vecteurs de dimension C"*,
ou mieux : *"A est une **matrice** de dimensions L et C"*.

2. Les tableaux à deux dimensions

2.1. Déclaration et mémorisation

Déclaration

```
<TypeSimple> <NomTabl>[<DimLigne>][<DimCol>];
```

Exemples

Les déclarations suivantes en langage algorithmique :

tableau entier A[10,10]
tableau réel B[2,20]
tableau booléen C[3,3]
tableau caractère D[15,40]

se laissent traduire en C par:

`int A[10][10];`	ou	`long A[10][10];`	ou	...
`Float B[2][20];`	ou	`double B[2][20];`	ou	...
`Int C[3][3];`				
`char D[15][40];`				

2. Les tableaux à deux dimensions

Mémorisation

Comme pour les tableaux à une dimension, le nom d'un tableau est le représentant de ***l'adresse du premier élément*** du tableau (c.-à-d. l'adresse de la première ***ligne*** du tableau).

Les composantes d'un tableau à deux dimensions sont stockées ligne par ligne dans la mémoire.

2. Les tableaux à deux dimensions

***Exemple*: Mémorisation d'un tableau à deux dimensions**

```
short A[3][2] = {{1, 2}, {10, 20}, {100, 200}};
```

Un tableau de dimensions L et C, formé de composantes dont chacune a besoin de M octets, occupera L*C*M octets en mémoire.

Exemple

En supposant qu'une variable du type **double** occupe 8 octets (càd : sizeof(double)=8),

pour le tableau T déclaré par:

```
double T[10][15];
```

C réservera L*C*M = 10*15*8 = 1200 octets en mémoire.

2. Les tableaux à deux dimensions

2.2. Initialisation et réservation automatique

Initialisation

Lors de la déclaration d'un tableau, on peut initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades.

A l'intérieur de la liste, les composantes de chaque ligne du tableau sont encore une fois comprises entre accolades. Pour améliorer la lisibilité des programmes, on peut indiquer les composantes dans plusieurs lignes.

2. Les tableaux à deux dimensions

Exemples

```
int A[3][10] ={ { 0,10,20,30,40,50,60,70,80,90},
                {10,11,12,13,14,15,16,17,18,19},
                {1,12,23,34,45,56,67,78,89,90} };
float B[3][2] = {{-1.05,-1.10},{86e-5,87e-5},{-12.5E4,-12.3E4}};
```

Lors de l'initialisation, les valeurs sont affectées ligne par ligne en passant de gauche à droite.

Nous ne devons pas nécessairement indiquer toutes les valeurs: Les valeurs manquantes seront initialisées par zéro.

Il est cependant défendu d'indiquer trop de valeurs pour un tableau.

2. Les tableaux à deux dimensions

Exemples

```
int C[4][4] = {{1, 0, 0, 0}
               {1, 1, 0, 0}
               {1, 1, 1, 0}
               {1, 1, 1, 1}};
```

C:

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

2. Les tableaux à deux dimensions

Exemples

```
int C[4][4] ={{1, 1, 1, 1}};
```

C:

1	1	1	1
0	0	0	0
0	0	0	0
0	0	0	0

```
int C[4][4] ={{1}, {1}, {1}, {1}};
```

C:

1	0	0	0
1	0	0	0
1	0	0	0
1	0	0	0

2. Les tableaux à deux dimensions

Réservation automatique

Si le nombre de **lignes L** n'est pas indiqué explicitement lors de l'initialisation, l'ordinateur réserve automatiquement le nombre d'octets nécessaires.

```
int A[][10] ={ {0,10,20,30,40,50,60,70,80,90},
               {10,11,12,13,14,15,16,17,18,19},
               {1,12,23,34,45,56,67,78,89,90} };
```

➔ réservation de 3*10*2 = 60 octets

```
float B[][2] = {{-1.05,-1.10},{86e-5,87e-5 },{-12.5E4,-12.3E4}};
```

➔ réservation de 3*2*4 = 24 octets

2. Les tableaux à deux dimensions

Exemple

```
int C[][4] = {{1, 0, 0, 0}
              {1, 1, 0, 0}
              {1, 1, 1, 0}
              {1, 1, 1, 1}};
```

C:

1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1

⇒ réservation de 4*4*2 = 32 octets

2. Les tableaux à deux dimensions

2.3. Accès aux composantes

Accès à un tableau à deux dimensions

<NomTableau>[<Ligne>][<Colonne>]

Les éléments d'un tableau de dimensions L et C se présentent de la façon suivante:

```
| A[0][0]   A[0][1]   A[0][2]  ...  A[0][C-1]    |
| A[1][0]   A[1][1]   A[1][2]  ...  A[1][C-1]    |
| A[2][0]   A[2][1]   A[2][2]  ...  A[2][C-1]    |
|   ...       ...       ...     ...   ...        |
| A[L-1][0] A[L-1][1] A[L-1][2] ... A[L-1][C-1]  |
```

2. Les tableaux à deux dimensions

Remarque

Considérons un tableau A de dimensions **L** et **C**.

En C

- Les indices du tableau varient de **0** à **L-1**, respectivement de **0** à **C-1**.
-La composante de la N^{ième} ligne et M^{ième} colonne est notée :

```
A[N-1][M-1]
```

En langage algorithmique

- Les indices du tableau varient de **1** à **L**, respectivement de **1** à **C**.
-La composante de la N^{ième} ligne et M^{ième} colonne est notée:

```
A[N,M]
```

Les Pointeurs

1. Adressage
    1.1. Adressage direct
    1.2. Adressage indirect

2. Les Pointeurs
    2.1. Les opérateurs de base
    2.2. Les opérations élémentaires sur pointeurs

Les Pointeurs

1. Adressage

L'importance des pointeurs en C

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de ***pointeurs***, c.-à-d. à l'aide de variables auxquelles on peut attribuer les ***adresses*** *d'autres variables*.

1. Adressage

1.1. Adressage direct

Dans la programmation, nous utilisons des variables pour stocker des informations.

La valeur d'une variable se trouve à un endroit spécifique dans la mémoire interne de l'ordinateur.

Le nom de la variable nous permet alors d'accéder *directement* à cette valeur.

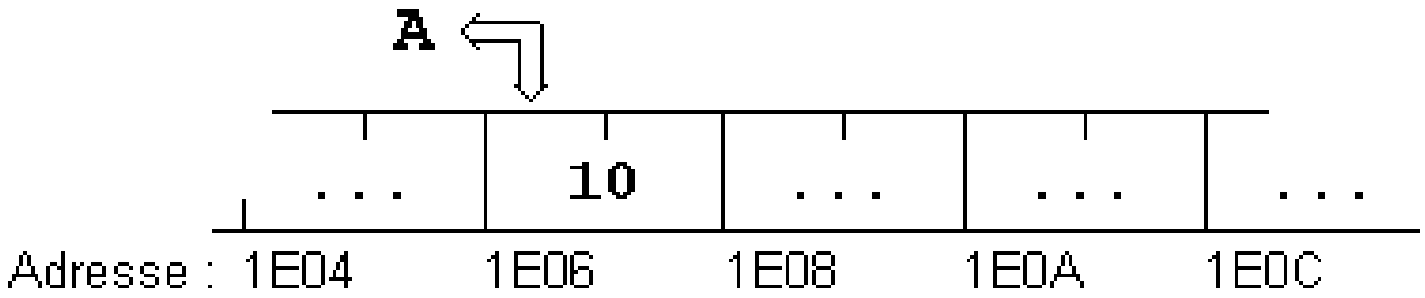
Adressage direct

Accès au contenu d'une variable par le nom de la variable.

1. Adressage

Exemple

```
short A;
A = 10;
```



1. Adressage

1.2. Adressage indirect

Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, *nous pouvons copier l'adresse de cette variable dans une variable spéciale P, appelée* ***pointeur*****.** Ensuite, nous pouvons retrouver l'information de la variable A en passant par le pointeur P.

Adressage indirect

Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

2. Les Pointeurs

Définition

*Un **pointeur** est une variable spéciale qui peut contenir l'**adresse** d'une autre variable.*

En C, chaque pointeur est limité à un type de données.

Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

Si un pointeur P contient l'adresse d'une variable A, on dit que :

'P pointe sur A'

2. Les Pointeurs

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur.

Il faut quand même bien faire la différence:

-Un ***pointeur*** est une variable qui peut 'pointer' sur différentes adresses.

- Le ***nom d'une variable*** reste toujours lié à la même adresse.

2. Les Pointeurs

2.1. Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur 'adresse de' : **&** pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de' : ***** pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration pour pouvoir déclarer un pointeur.

2. Les Pointeurs

L'opérateur 'adresse de' : **`&`**

`&<NomVariable>` fournit l'adresse de la variable **`<NomVariable>`**
L'opérateur **`&`** nous est déjà familier par la fonction **`scanf`**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Par exemple : **`scanf("%d", &N);`**

Remarque
L'opérateur **`&`** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

2. Les Pointeurs

Représentation schématique

Soit P un pointeur non initialisé

et A une variable contenant la valeur 10.

Alors l'instruction

```
P = &A;
```

affecte l'adresse de la variable A à la variable P.

2. Les Pointeurs

L'opérateur 'contenu de' : `*`

`*<NomPointeur>` désigne le contenu de l'adresse référencée par le pointeur **`<NomPointeur>`**

Exemple

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé :

Après les instructions,

```
P = &A;    B = *P;    *P = 20;
```

P pointe sur A,
Le contenu de A (référencé par *P) est affecté à B et mis à 20.

2. Les Pointeurs

Déclaration d'un pointeur

`<Type> *<NomPointeur>` déclare un pointeur **`<NomPointeur>`** qui peut recevoir des adresses de variables du type **`<Type>`**

Une déclaration comme `int *PNUM;`
peut être interprétée comme suit:

	PNUM est du type **int
Ou	
	*PNUM est un pointeur sur **int***
Ou	
	*PNUM peut contenir l'adresse d'une variable du type **int***

2. Les Pointeurs

Exemple

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit:

`Main()`	ou	`main()`
`{`		`{`
`/* déclarations */`		`/* déclarations */`
`Short A = 10;`		`short A, B, *P;`
`Short B = 50;`		`/* traitement */`
`Short *P;`		`A = 10;`
`/* traitement */`		`B = 50;`
`P = &A;`		`P = &A;`
`B = *P;`		`B = *P;`
`*P = 20;`		`*P = 20;`
`Return 0;`		`return 0;`
`}`		`}`

2. Les Pointeurs

Remarque

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données.

Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.