“ Algorithmique avancée ”

1- Introduction

Pr. Abdelhay HAQIQ (ahaqiq@esi.ac.ma)

Plan

- ❑ **Rappel des notions de base**
- ❑ **Récursivité**
- ❑ **Complexité**

Plan

- ❑ **Rappel des notions de base**
- ❑ Récursivité
- ❑ Complexité

Opérateurs

- Les opérateurs sont utilisés pour effectuer des opérations sur des variables et des valeurs.
  - incrémentation/décrémentation (++, --)
  - arithmétiques (+, -, *, =, %)
  - relationnels (>, <, >=, <=, ==, ! =)
  - logique (!, &&, | |)
  - affectation (=, +=, -=,*=, /=, %=)

Exercice 1

- Quels résultats fournit ce programme ?

```
int i, j, n;
i = 0;
n = i++;
printf("A : i = %d et n = %d \n", i, n);
i = 10;
n = ++i;
printf("B : i = %d et n = %d \n", i, n);
```

Quels résultats fournit ce programme ?

```c
int i, j, n;
i = 20;
j = 5;
n = i++ * ++j;
printf("C : i = %d j = %d n = %d\n", i, j, n);
i = 15;
n = i += 3;
printf("D : i = %d n = %d\n", i, n);
i = 3;
j = 5;
n = i *= --j;
printf("E : i = %d j = %d n = %d\n", i, j, n);
```


Exercice 2

Structures de contrôle

- Les structures de contrôles sont de trois types :
  - Séquence : exécution séquentielle d'une suite d'instructions séparées par un point-virgule
  - Alternative : structure permettant un choix entre divers blocs d'instructions suivant le résultat d'un test logique
  - Boucle : structure itérative permettant de répéter plusieurs fois la même bloc d'instructions tant qu'une condition de sortie n'est pas avérée

Structure Alternative : If/Else

IF – ELSE

```
if (condition)
{
   //instructions
}
else
{
   //instructions    dans
  les autres cas
}
```

IF – ELSE – IF – ELSE

```
if (condition)
{
   //instructions
}
else
   if (condition)
   {
      //instructions
   }
   else
   {
     //instructions
   }
```

Quels résultats fournit ce programme ?

```c
double n1 = -1.0, n2 = 4.5, n3 = -5.3, nombre;
    if (n1 >= n2)
    {
        if (n1 >= n3)
        {
            nombre = n1;
        }
        else
        {
            nombre = n3;
        }
    }
    else
    {
        if (n2 >= n3)
        {
            nombre = n2;
        }
        else
        {
            nombre = n3;
        }
    }
    printf("Le nombre est : %lf ", nombre);
```

Structure Alternative : If/Else

Structure Alternative : If/Else

```
    if (x == 0)
    {
        instruction1;
    }
    else
    {
        if (x == 1)
        {
            instruction2;
        }
        else
        {
            if (x == 2)
            {
                instruction3;
            }
            else
            {
                if (x == 3)
                {
                    instruction4;
                }
                else
                {
                    /* instruction à exécuter par défaut, au cas où x n'égale
                    aucune des valeurs 0,1,2 ou 3 */
                    instruction5;
                }
            }
        }
    }
}
```

Problème : Plusieurs condition IF à gérer
Solution : Switch - case

Structure Alternative : SWITCH-CASE

```
switch(expression)
    {
    case x:
        // instructions
        break;
    case y:
        // instructions
        break;
    default:
        // instructions
    }
```

Structure Alternative : SWITCH-CASE

Exercice 1

```c
int n;
printf("Donnez une valeur:  ");
scanf("%d", &n) ;

switch (n) {
        case 0:
          printf("Nul\n");
        case 1:
        case 2:
          printf("Petit\n");
          break;
        case 3:
        case 4:
        case 5:
         printf("Moyen\n");
        default:
         printf("Grand\n");
      }
```

Quels résultats affiche-t-il lorsqu'on lui fournit en donnée :

- la valeur 0
- la valeur 1
- la valeur 4
- la valeur 10
- la valeur -5

Exercice 2

Écrire un programme qui lit le nombre d'enfants d'une famille, et qui affiche le montant de l'allocation familiale que doit recevoir cette famille, selon les règles suivantes :

- si la famille ne contient pas d'enfants, aucune allocation
- entre 1 et 3 enfants, allocation de 150dh
- entre 4 et 6 enfants, allocation de 250dh
- plus de 7 enfants, allocation de 350dh
- si le nombre d'enfants est incorrecte, afficher un message d'erreur.

Boucle

```
for (instructionInit; condition; instructionIter)  instruction;

for (instructionInit; condition; instructionIter) {
  instruction1;
  instruction2;
  ...
}
```

```
while (condition) instruction;


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

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

Exercice

Écrire un programme qui lit un entier N et qui affiche les entiers de 1 à N, 5 par 5, séparés par des tabulations

Exemple d'exécution pour N = 22 :

1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

16 17 18 19 20

21 22

<u>*Indication*</u> : Utiliser l'opérateur *modulo* (%)

Break / Continue

- Certaines instructions permettent un contrôle supplémentaire sur les boucles :
  - **BREAK** : permet de quitter immédiatement une boucle ou un branchement
  - **CONTINUE** : permet d'ignorer le reste des instructions et de passer directement à l'itération suivante

```
for (i=0; i<10 ; i++)
{
    if (i==5) continue; // Si i=5, on passe à l'itération suivante
    if (i==7) break;      // Si i=7, on sort de la boucle
    printf("La valeur de i est : %d\n", i);
}
```

Résultat est :

La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
La valeur de i est : 6

Quels résultats fournit ce programme ?

```c
int n = 0;
do
{
    if (n % 2 == 0)
    {
        printf("%d est pair\n", n);
        n += 3;
        continue;
    }
    if (n % 3 == 0)
    {
        printf("%d est multiple de 3\n", n);
        n += 5;
    }
    if (n % 5 == 0)
    {
        printf("%d est multiple de 5\n", n);
        break;
    }
    n += 1;
} while (1);
```

Pointeurs

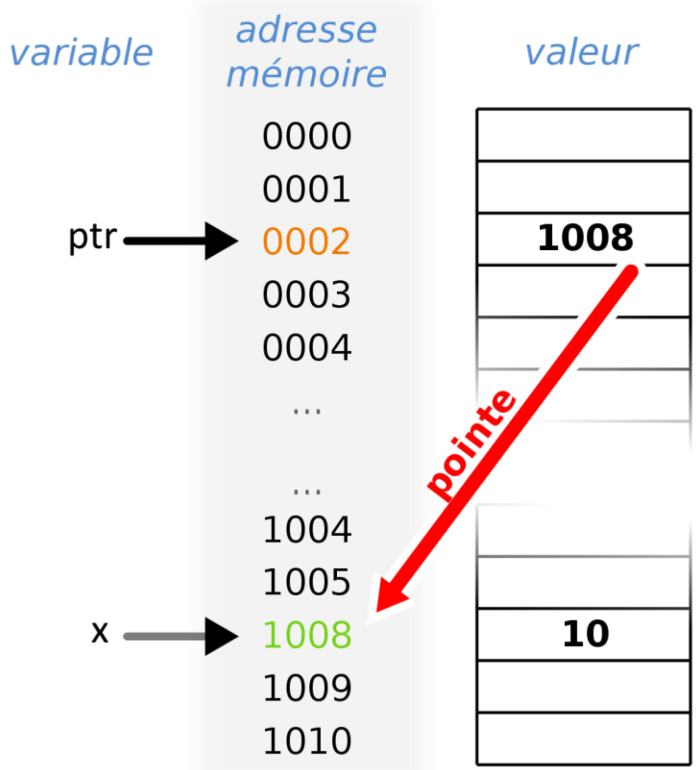
- Un **pointeur** est une variable qui qui contient l'adresse d'une autre variable

- La déclaration : Type_donnee * pointeur;
  - Exemple :

    int x = 10;

    Int * ptr = &x;

    // ptr est un variable qui stocke l'adresse de x



Exercice 1

Quels résultats fournit ce programme ?

```c
int x = 43;
int* ptr = &x;

printf("%d\n", x);

printf("%p\n", &x);

printf("%p\n", &ptr);

printf("%p\n", ptr);

printf("%d\n", *ptr);

*ptr = 20;

printf("%d\n", *ptr);

printf("%d\n", x);
```

Exercice 2

Quels résultats fournit ce programme ?

```c
void permuter(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}

 int main()
 {
   int a = 10;
   int b = 20;

 printf("Avant :\n la valeur de a : %d \n la valeur de b : %d\n", a, b);

 permuter(a, b);

 printf("Apres :\n la valeur de a : %d \n la valeur de b : %d\n", a, b);
 return 0;
 }
```

Exercice 3

Quels résultats fournit ce programme ?

```c
void permuter(int *x, int *y)
{
  int temp;

  temp = *x;
  *x = *y;
  *y = temp;
}

 int main()
 {
   int a = 10;
   int b = 20;

 printf("Avant :\n la valeur de a : %d \n la valeur de b : %d\n", a, b);

 permuter(&a, &b);

 printf("Apres :\n la valeur de a : %d \n la valeur de b : %d\n", a, b);
 return 0;
 }
```

- Lors de l'appel, les adresses de a et de b sont copiées dans les pointeurs x et y.
- PERMUTER échange ensuite le contenu des adresses indiquées par les pointeurs x et y.

Tableaux

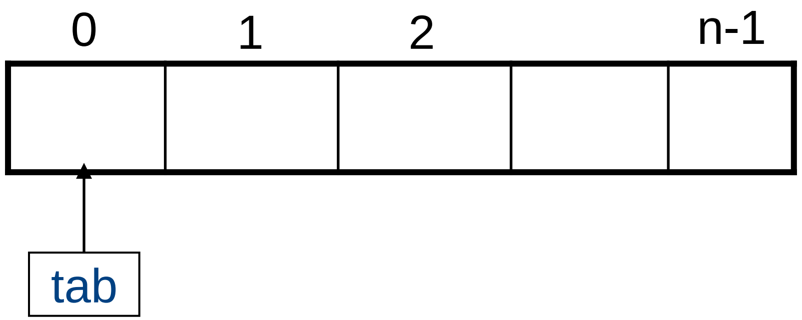
- Un tableau est un ensemble d'éléments de même type désignés par un identificateur unique
- On définit les tableaux de la façon suivante :
  - Exemple : int tabEntier [20]; //définit un tableau de 20 entiers
  - Exemple : char tabCaractere[ ] = {'a', 'l', 'l', 'o', '\0'}; //définit un tableau de 4 caractères

- Forme générale :
  - type [ ] nom_tableau = { liste des valeurs séparées par une virgule };
    - int t1 [] = {1, 3, 5} ;
    - t1 [2] = 78;
  - type * nom_tableau = malloc(taille * sizeof(type)); //ajouter la bibliothèque `#include <stdlib.h>`

Tableaux

- L'identificateur d'un tableau, lorsqu'il est employé seul (sans indices à sa suite), est considéré comme un pointeur (constant) sur le début du tableau

Pointeur			Tableau
	`tab+i`	⇔	`&tab[i]`
	`*(tab+i)`	⇔	`tab[i]`



- Supposons `int t[10];`
- Les notations suivantes sont équivalentes:

`t`	`&t[0]`
`t+1`	`&t[1]`
`t+i`	`&t[i]`
`t[i]`	`*(t+i)`

```
printf("%d",*MyTab); ou printf("%d",MyTab[0]);
scanf("%d",MyTab+2); ou scanf("%d",&MyTab[2]);
```

Exercice 1

- Écrire un programme qui crée un tableau comportant les valeurs des carrés des n nombres impairs, la valeur de n étant lue au clavier et qui en affiche les valeurs sous la forme suivante :
- Combien de valeurs : 5
  - 1 a pour carre 1
  - 3 a pour carre 9
  - 5 a pour carre 25
  - 7 a pour carre 49
  - 9 a pour carre 81

Plan

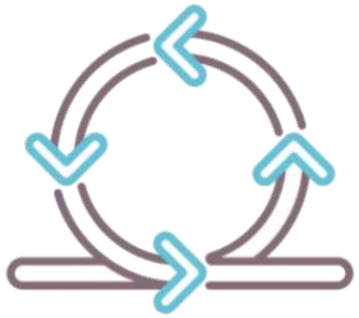
- Rappel des notions de base
- **Récursivité**
- Complexité

Solution itérative du factoriel

```
Fonction factoriel ( N : Entier ) : Entier
    Variables
        F , i : Entier
    Début
        Si N = 0 ou N = 1 Alors
            F ← 1
        Sinon
            F ← 1
            Pour i ← 2 à N pas 1 Faire
                F ← i * F
            Fin Pour
        Fin si
        Retourne F
    Fin
```

```
    ...
    Ecrire ( " Le factoriel du nombre " , N , " est : " , factoriel ( N ) )
Fin
```

Solution itérative



Algorithme factoriel

factoriel (1) = 1

factoriel (2) = 2 * factoriel (1)

factoriel (3) = 3 * factoriel (2)

factoriel (4) = 4 * factoriel (3)

factoriel (5) = 5 * factoriel (4)

factoriel (N) = N * factoriel (N - 1)

Récursivité

- Une fonction **récursive** est une fonction qui s'appelle elle-même
- Elle est définie par :
  - Au moins un cas de base
  - Au moins un cas récursif (cas général)

Solution récursive du factoriel

Cas de base

$$\text{factoriel}(N) = \begin{cases} 1 & \text{Si } N = 0 \text{ ou } N = 1 \\ N * \text{factoriel}(N - 1) & \text{Si } N > 1 \end{cases}$$

Cas récursif

Solution récursive du factoriel

```
Fonction factoriel ( N : Entier ) : Entier
        Début
                Si  N = 0 ou N = 1 Alors
                        Retourne 1
                Sinon
                        Retourne N * factoriel ( N – 1 )
                Fin si
        Fin
```

```
        ...
        Ecrire ( " Le factoriel du nombre " , 5 , " est : " , factoriel ( 5 ) )
Fin
```

Notion de pile d’exécution

- La **Pile d’exécution** (call stack) du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que l’adresse de retour de chaque fonction en cours d’exécution
- Elle fonctionne selon le principe LIFO (Last-In-First-Out)

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (**stack overflow**)

Notion de pile d'exécution

```
Fonction factoriel ( N : Entier ) : Entier
    Début
        Si  N = 0 ou N = 1 Alors
                Retourne 1
        Sinon
                Retourne N * factoriel ( N – 1 )
        Fin si
    Fin
```

```
    ...
    Ecrire ( " Le factoriel du nombre " , 5 , " est : " , factoriel ( 5 ) )
Fin
```

RAM	RAM
Factoriel(1)	1
Factoriel(2)	2 = 2 x Factoriel(1)
Factoriel(3)	6 = 3 x Factoriel(2)
Factoriel(4)	24 = 4 x Factoriel(3)
Factoriel(5)	120 = 5 x Factoriel(4)

Exercice 1

- Ecrire un algorithme qui demande à l'utilisateur de taper un entier positif n. Ensuite, à l'aide d'une fonction récursive, l'algorithme calcule et affiche tous les termes de la suite de Fibonacci, inférieurs ou égaux à n.
- La suite de Fibonacci est définie comme suit :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_n = U_{n-1} + U_{n-2} \quad (\text{n}>1) \end{cases}$$

Exercice 2

- Ecrire un programme qui demande à l'utilisateur de taper un entier positif n. Ensuite, à l'aide d'une fonction récursive, l'algorithme calcule la somme des nombres de 1 à n.

Exercice 3

- Ecrire un programme récursif qui calcule la somme des éléments positifs d'un tableau.
- Deux paramètres : un tableau d'entiers tab, une taille n et un indice i. Le but de la fonction est de renvoyer la somme des entiers positifs du tableau compris entre i et n.

Plan

- ❑ Rappel des notions de base
- ❑ Récursivité
- ❑ **Complexité**

Complexité

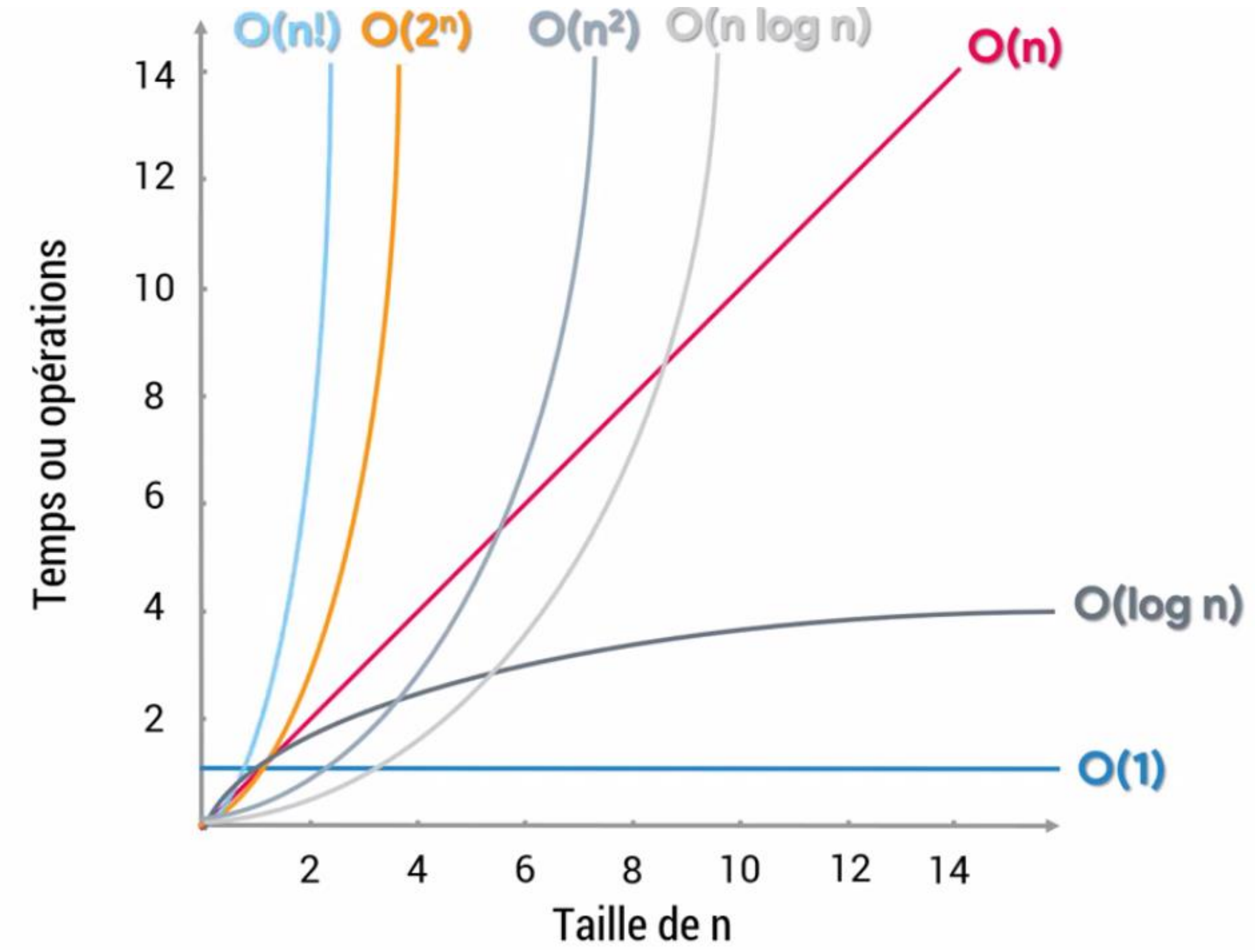
- La complexité d'un algorithme quantifie le **temps nécessaire** à un algorithme **pour s'exécuter** en fonction de la **taille de l'entrée**
- La **complexité** d'un algorithme consiste en l'étude de la quantité de ressources (de **temps** ou **d'espace**) nécessaire à l'exécution de cet algorithme
  - La **complexité temporelle** d'un algorithme quantifié le **temps** nécessaire à un algorithme pour s'exécuter en fonction de la longueur de l'entrée
  - La **complexité spatiale** d'un algorithme quantifier la quantité **d'espace** ou de **mémoire** prise par un algorithme pour s'exécuté en fonction de la longueur de l'entrée

Complexité

- Pour un problème donné, il existe **plusieurs algorithmes** qui résolvent ce problème. Pour choisir le meilleur, nous devons comparer et analyser les performances de chaque algorithme
- Lors de l'analyse d'un algorithme, nous considérons principalement l'étude de la complexité
- La notation Grand **O** est une métrique permettant de décrire le temps d'exécution d'un algorithme

Types de complexité

Ordre de grandeur du temps nécessaire à l'exécution d'un algorithme d'un type de complexité										
	Temps	Type de complexité	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
Complexité polynomiale (P)	O(1)	Complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	Accès à une cellule de tableau
	O(log(n))	Complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	Recherche dichotomique
	O(n)	Complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	Parcours d'une liste
	O(nlog(n))	Complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	Tris par comparaisons optimaux (comme le tri fusion)
	O(n^2)	Complexité quadratique (polynomiale)	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	Parcours de tableaux 2D
	O(n^3)	Complexité cubique (polynomiale)	1.25 µs	10 µs	80 µs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	Multiplication matricielle
Complexité non polynomiale (NP)	O($2^{poly(n)}$)	Complexité exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans	...	...	...	Problème du sac à dos
	O(n!)	Complexité factorielle	1.2 µs	36 ms	770 ans	10^{48} ans	...	...	...	...	Problème du voyageur de commerce



Types de complexité

Règles de calcul de la complexité

- Pour calculer la complexité Grand O d'un algorithme il faut compter le nombre **d'opération de base** qu'il effectue comme :
  - Opération **arithmétique** ou logique (+, * et , ou, ..)
  - Opération **d'affectation** (x = 10)
  - Vérification **d'une condition** (x> 10)
  - Opération **d'entrée/sortie** (Ecrire ou Lire)
- La complexité de chaque opération de base est **constante** ou **O(1)**

Règles de calcul de la complexité

- La complexité d'une **boucle** est la complexité du bloc interne dans la **boucle multipliée** par le nombre de fois que le bloc interne est répété
- Exemple :

```
for (i = 1; i <= n; i++)
 {
     printf("Entrer un nombre : ");

     scanf("%d", &x);

     printf("i x a = %d", x * i);
 }
```

O(1)

O(1)

O(1)

O(1)

O(1)

x n

La complexité est O(5n)

Règles de calcul de la complexité

- La complexité de la structure **Si/Sinon** correspond à la complexité de la condition O(1) plus la complexité la plus grande entre « If » et « else »
- Exemple

```
if (n < 10)                                        ← O(1)
   {
       printf("La valeur est inférieure à 10");    ← O(1)
   }
else
   {
       for (i = 1; i <= n; i++)                    ← O(1)  ┐
       {                                                   ├ O(2n)
           printf("La valeur de i est %d: ", i);   ← O(1)  ┘
       }
   }
```

La complexité est :

O(1) + Max(O(1), O(2n))

= O(2n + 1)

Règles de calcul de la complexité

- La complexité **d'une séquence de deux blocs** d'instructions est égale à la plus grande des complexités des deux blocs

```c
for (i = 1; i <= n; i++)
 {
      printf("Entrer un nombre : ");
      scanf("%d", &x);
      printf("i x a = %d", x * i);
 }
```

```c
if (n < 10)
     printf("La valeur est inférieure à 10");
else
   {
      for (i = 1; i <= n; i++)
      {
          printf("La valeur de i est %d: ", i);
      }
   }
```

La complexité est :

Max(O(5n), O(2n+1)) = O(5n)

Règles de calcul de la complexité

- La complexité d'un algorithme est un calcul de ses performances **asymptotiques** dans le **pire des cas,**
- **Asymptotique**, nous nous intéressons aux données très volumineuses car les petites valeurs ne sont pas assez informative
  - Les constantes **multiplicatives** sont remplacées par 1
  - Les constantes **additives** sont annulées
  - Le terme le **plus élevé** est conservé
- Exemple : $O(6n^2 + 7n + 4)$ ➡ $O(n^2)$

Exercice 1

```
Algorithme signe
Variables
    n : Entier
Début
    Ecrire ( " Veuillez entrer un nombre : " )
    Lire ( n )
    Si  n > 0 Alors
        Ecrire ( " Ce nombre est positif ")
    Sinon
        Ecrire ( " Ce nombre est négatif " )
    Fin si
Fin
```

La complexité est : Max(O(1), O(1), O(2)) = O(2)

Donc la complexité est constante O(C)= O(1)

Exercice 2

```
Algorithme affichage
Variables
    n , i : Entier
Début
    Ecrire ( " Veuillez entrer un nombre : " )
    Lire ( n )
    Pour i ← n+1 à n+10 pas 1 Faire
        Ecrire (i)
    Fin Pour
Fin
```

La complexité est : Max(O(1), O(1), O(20)) = O(20)

Donc la complexité est constante O(C)= O(1)

Exercice 3

```
Algorithme minimum
Variables
        Tableau T ( ) : entier
        i , n : entier
        Min : réel
Début
        Ecrire ( " Veuillez saisir la taille du tableau : " )
        Lire ( n )
        Ecrire ( " Veuillez saisir les éléments du tableau : " )
        Pour i ← 0 à n - 1 pas 1 Faire
                Lire ( T ( i ) )
        Fin Pour
        Min ← T (0)
        Pour i ← 1 à n - 1 pas 1 Faire
                Si Min > T (i) alors
                        Min ← T (i)
                fin Si
        Fin Pour
        Ecrire ( " Le minimum des éléments est : " , Min )
Fin
```

La complexité est :

Max(O(1), O(1), O(1), O(2n), O(1), O(3n), O(1)) = O(3n)

Donc la complexité est linéaire O(C)= O(n)

Exercice 4

```
Algorithme minimum
Variables
          Tableau T ( , ) : entier
          i , j , n : entier

Début
          Ecrire ( " Veuillez saisir la taille de la matrice carré : " )
          Lire ( n )
          Ecrire ( " Veuillez saisir les éléments de la matrice : " )
          Pour i ← 0 à n - 1 pas 1 Faire
                    Pour j ← 0 à n - 1 pas 1 Faire
                              Lire ( T ( i , j ) )
                    Fin Pour
          Fin Pour
          Ecrire ( " Affichage des éléments de la matrice : " )
          Pour i ← 0 à n - 1 pas 1 Faire
                    Pour j ← 0 à n - 1 pas 1 Faire
                              Lire ( T ( i , j ) )
                    Fin Pour
          Fin Pour
Fin
```

La complexité est quadratique : $O(n^2)$

```
Algorithme affichage
Variables
    n , i : Entier
Début
    Ecrire ( " Veuillez entrer un nombre : " )
    Lire ( n )
    Pour i ← 1 à n pas i*2 Faire
        Ecrire (i)
    Fin Pour
Fin
```

x $1+\log_2(n)$

Exercice 5

La complexité est logarithmique : $O(\log_2(n))$

Exercice 6

```
...
Ecrire ( " Veuillez saisir un nombre: " )
Lire ( n )
Pour i ← 1 à n pas 1 Faire
        Pour j ← 1 à n pas j*4 Faire
                Ecrire ( i )
                Ecrire ( j )
        Fin Pour
Fin Pour
...
```

x n

x $\log_4(n)$

$1 + \log_4(10)$

$1 + 1{,}6 \approx 2$

Si n vaut 300

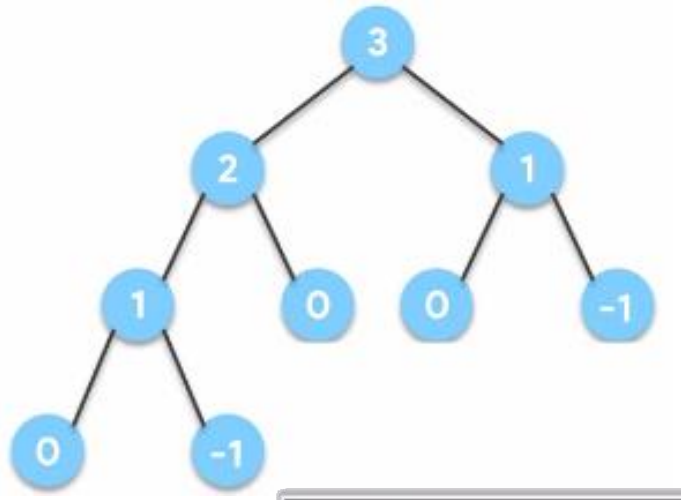
$1 + \log_4(300)$

$1 + 4{,}1 \approx 5$

La complexité est : Max(O(1), O($n\log_4(n)$))

Donc la complexité est linéarithmique O(C)= O($n\log_4(n)$)

Si n vaut 1 ➡ 3 appels	Si n vaut 2 ➡ 5 appels
Si n vaut 3 ➡ 9 appels	Si n vaut 4 ➡ 15 appels



Exercice 7

```
Fonction fn ( n : Entier ) : Entier
    Début
        Si  n <= 0 Alors
            Retourne 1
        Fin Si
        Retourne fn ( n – 1 ) + fn ( n – 2 )
    Fin
```

$2^1 + 1 = 3$

$2^2 + 1 = 5$

$2^3 + 1 = 9$

$2^4 - 1 = 15$

$2^5 - 7 = 25$

$2^n + c$

La complexité est exponentielle O(C)= O(2^n)