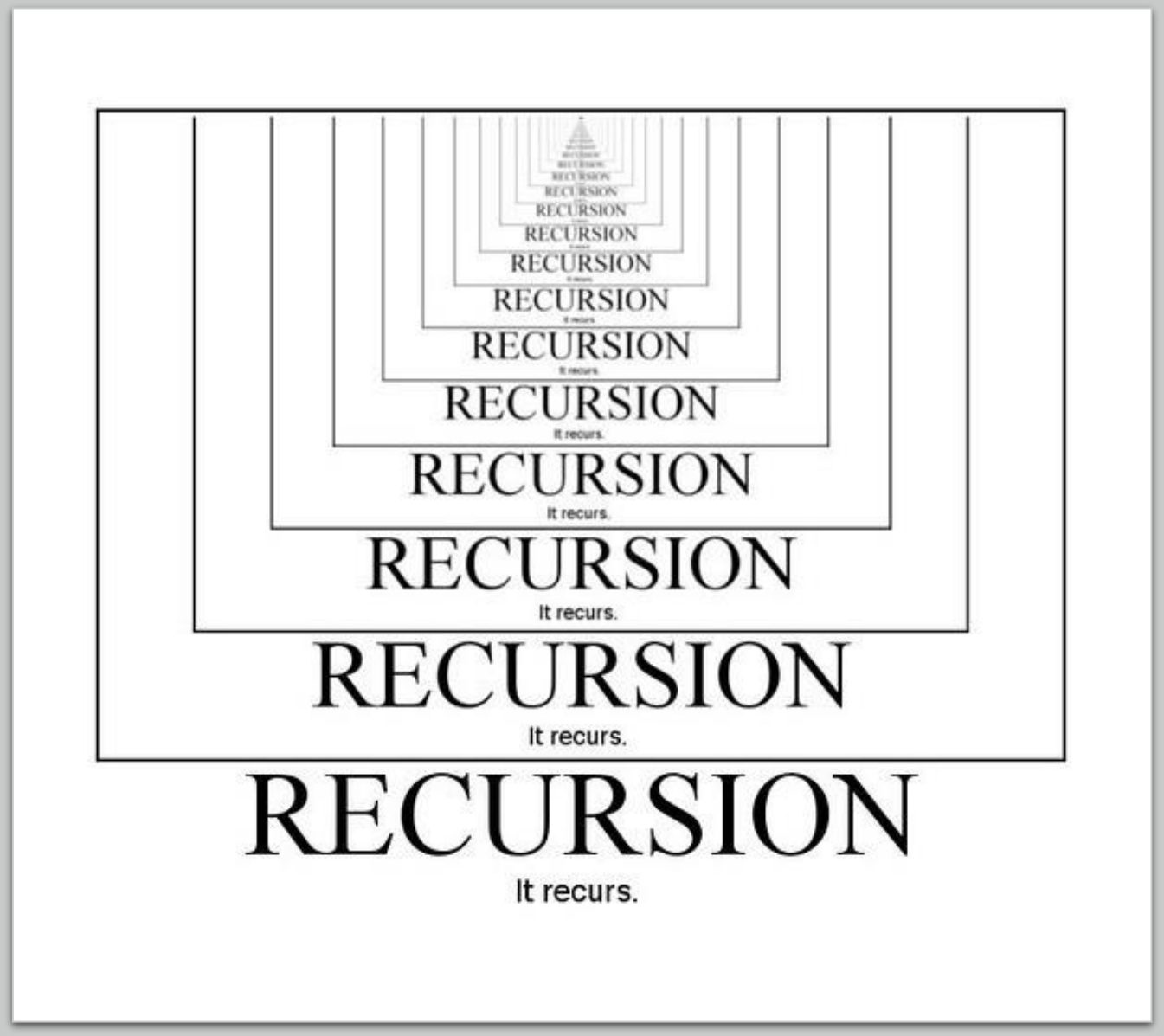
# Algorithmique & Structures de Données

**L2 Informatique 2024-2025**

## *Chapitre 2 :*

# *La RÉCURSIVITÉ*



# Définitions

La récursivité est un des concepts de base en informatique, qui peut être illustré (quasiment) dans tous les langages de programmation et qui peut être utile dans de nombreuses situations.

Un objet est récursif s'il est utilisé directement (ou indirectement) dans sa définition. Un algorithme est dit récursif lorsqu'il s'appelle lui même.

Principe : est d'utiliser pour décrire l'algorithme sur un domaine D, l'algorithme lui-même à un sous ensemble D' (différent de D).

# Exemple1: Factorielle

$5 ! = 5 \times 4 \times 3 \times 2 \times 1$

Factorial (n) = $n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$

Forme iterative :

```
int factorial(int n)
 {
    int f = 1;
    for ( int i=1; i<=n; i++ )
    f = f*i;
    return f;
  }
```

Factorial (n) = n × (n-1) × (n-2) × … ×3×2×1

**Factorial (n) = n × Factorial (n-1)**

On peut écrire : $Factorial(n) = \begin{cases} 1 & if \quad n \leq 1 \\ n \times Factorial(n-1) & if \quad n > 1 \end{cases}$

Forme récursive :

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
     else
        return n * factorial(n - 1);
}
```

# Pendant l'exécution . . .

```
...
f = factorial(4);
...
```

```
int factorial(int 4)
  if (4 <= 1) return 1;
  else return 4 * factorial( 3 );
```

```
int factorial(int 3)
  if (3 <= 1) return 1;
  else return 3 * factorial( 2 );
```

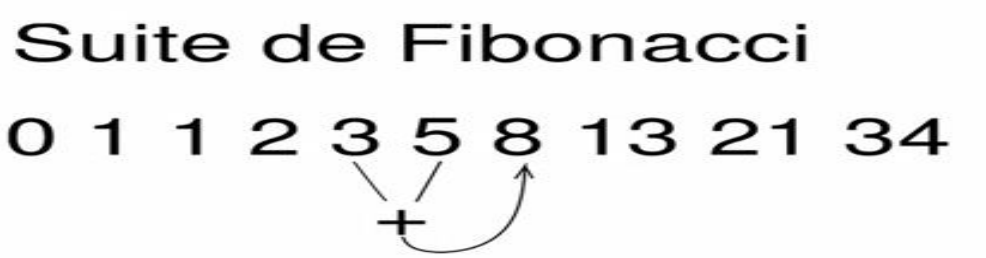
```
int factorial(int 2)
  if (2 <= 1) return 1;
  else return 2 * factorial( 1 );
```

```
int factorial(int 1)
  if (1 <= 1) return 1;
  else return 1 * factorial(n-1);
```

- **Chaque fois qu'une fonction est appelée, une nouvelle instance de la fonction est créée. Chaque fois qu'une fonction " renvoie ", son instance est détruite.**
- **Les instances d'une fonction sont détruites dans l'ordre inverse de leur création, c'est-à-dire que la première instance créée sera la dernière à être détruite.**

**Arrêt**

# Exemple2:




## Forme Iterative

```
int fib(int n)
{
  int a = 0, b = 1, c, i;
  if( n == 0)
    return a;
  for (i = 2; i <= n; i++)
   {
     c = a + b;
     a = b;
     b = c;
   }
  return b;
}
```

## Forme Récursive

```
int fib(int n)
{
    if (n < 2)
    return n;
    else
    return fib(n-1) + fib(n-2);
 }
```

# Exemple3: Afficher le contenu d'un tableau T de taille n

## Version récursive 1

```
void fct_display(int i, int T[], int n)

    {
        if (i<=n) {
                    printf("%d \n", T[i] );
                    fct_display( i+1, T, n);
                   }

    }
//premier appel
fct_display(0, T, n-1);
```

# Exemple3: Afficher le contenu d'un tableau T de taille n

## Version récursive 2

```
void fct_display(int T[],int n)
{
  if( n>=0 ) {
     fct_display( T, n-1);
     printf("%d \n", T[n] );
                        }
 }
//premier appel
fct_display(T, n-1);
```

## Version récursive 3

```
void fct_display(int T[],int n)
{
  if( n>=0 ) {
     printf("%d \n", T[n] );
     fct_display( T, n-1);
                        }
 }
//premier appel
fct_display(T, n-1);
```

# Exemple4: Recherche dichotomique

## Version itérative

```
int Iterative-Binary-Search(int A[],int v, int low, int high)
   {  while (low <= high){
                    mid =(low + high)/2;
                    if (v == A[mid]) return mid ;
                    if (v > A[mid])
                                  low = mid + 1 ;
                    else          high= mid - 1 ;
                    }
   return -1 ; }
```

# Exemple4: Recherche dichotomique

## Version récursive

```
int Recursive-Binary-Search(int A[],int v,int low,int high)
   { if (low > high)
                  return -1;
     mid = (low + high)/2;
     if (v == A[mid]) return mid ;
     if (v > A[mid])
        return Recursive-Binary-Search(A, v, mid+1, high);
   else return Recursive-Binary-Search(A, v, low, mid-1);
 }
```

# Exemple5: Vérifier si la liste est triée ?

**Version Itérative**

```
int arraySortedOrNot (int arr[], int n)
{
      if (n == 0 || n == 1)    return 1;
      for (int i = 1; i < n; i++)
            if (arr[i - 1] > arr[i])    return 0;
   return 1;
}
```

# Exemple5: Vérifier si la liste est triée ?

**Version récursive 1**

```
int arraySortedOrNot (int arr[], int n)

{

    if (n == 1 || n == 0)    return 1;

    if (arr[n - 1] < arr[n - 2])   return 0;

    return arraySortedOrNot (arr, n - 1);

}
```

# Exemple5: Vérifier si la liste est triée ?

**Version récursive 2**

```
int arraySortedOrNot(int A[], int n)

   {

       if (n == 1 || n == 0)     return 1;

      return A[n-1] > = A[n-2] && arraySortedOrNot(A,n-1);

   }
```

# Exemple6: Plus grand élément d'une liste ?

**Version itérative**

```
int max_array(int T[], int n)
{
    int i , big = T[0];
     for(i=1; i < n;i++)
          {
               if (T[i] > big )
                              big = T[i];
           }
       return big;
     }
```

# Exemple6: Plus grand élément d'une liste ?

## Version récursive

```
int largest (int A[], int position, int big)
    {
        if (position == 0)  return big;
        if (position > 0)
            {
             if (A[position] > big)
                        big = A[position];
             return largest (A, position - 1, big);
            }
    }
```

```
printf("The largest element in array: %d\n", largest(array, N-1, array[0]));
```

# Exemple7: Quel est l'affichage produit par l'exécution du programme ci-dessous:

```
int main()
    {
       printf("Début");
       compte (6);
       printf("FIN !");
    return 0 ; }
```

```
void compte ( int n  )
  {
    if(n < 12) {
       printf("%d\n ",n-1);
       compte (n+2);
       printf("%d\n ",n+1);
                  }
  }
```

```
Début
5
7
9
11
9
7
FIN !
```

# Exemple8: Vérifier si les éléments d'un Tableau sont tous distincts :

## Version Itérative

```
 int  distinctValues(int tab[], int n)
{
    for (int i = 0 ; i < n-1; i++)
          for (int j = i+1; j < n; j++)
              if (tab[i] == tab[j])  return 0;
    return 1;
}
```

# Exemple8: Vérifier si les éléments d'un Tableau sont tous distincts :

## Version Récursive

```
int distinctValues_rec(int tab[], int i, int j, int n)

{

   if(i==n) return 1; // cas d'arret

   if(j==n) return distinctValues_rec(tab, i+1, i+2, n) ;

   if(tab[i]==tab[j]) return 0;

   return distinctValues_rec(tab, i , j+1, n) ;

}
  // Premier Appel: int verif = distinctValues_rec ( T, 0, 1, n));
```

# Remarques :

- Identifier « **le cas d'arrêt** », *c'est à dire* **sans récursivité** (exemple : au moins 1 sortie pour une fonction de calcul ou 2 sorties (true or false) pour une fonction de vérification).
- Il est important de signaler que le nombre de niveaux (profondeur de la récursivité) doit être **fini** quelles que soient les valeurs des paramètres d'appel au départ, autrement le programme boucle indéfiniment.
- Certains langages de programmation n'admettent pas la récursivité (fortran, cobol, …)
- La récursivité est souvent coûteuse en temps et en espace mémoire.

# Formes de Récursivité :

## 1. Récursivité Simple :

$$x^n = \begin{cases} 1 & if\ n = 0 \\ x\ *\ x^{n-1} & if\ n \geq 1 \end{cases}$$

## 2. Récursivité Multiple :

Une définition récursive peut avoir plus d'un appel récursif.

$$C_n^p = \begin{cases} 1 & if\ p = 0\ \ or\ \ p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & else \end{cases}$$

# Formes de Récursivité :

## 3. Récursivité Mutuelle :

Les définitions sont dites mutuellement récursives si elles dépendent les unes des autres.

$$even(n) = \begin{cases} True & if\ n = 0 \\ odd(n-1) & else \end{cases}$$

$$odd(n) = \begin{cases} False & if\ n = 0 \\ even(n-1) & else \end{cases}$$

## 4. Récursivité imbriquée :

La fonction d'Ackerman est définie comme suit :

$$Acker(m,n) = \begin{cases} n+1 & if\ m = 0 \\ Acker(m-1,n) & if\ (m > 0)\ and\ (n = 0) \\ Acker(m-1, Acker(m,n-1)) & else \end{cases}$$