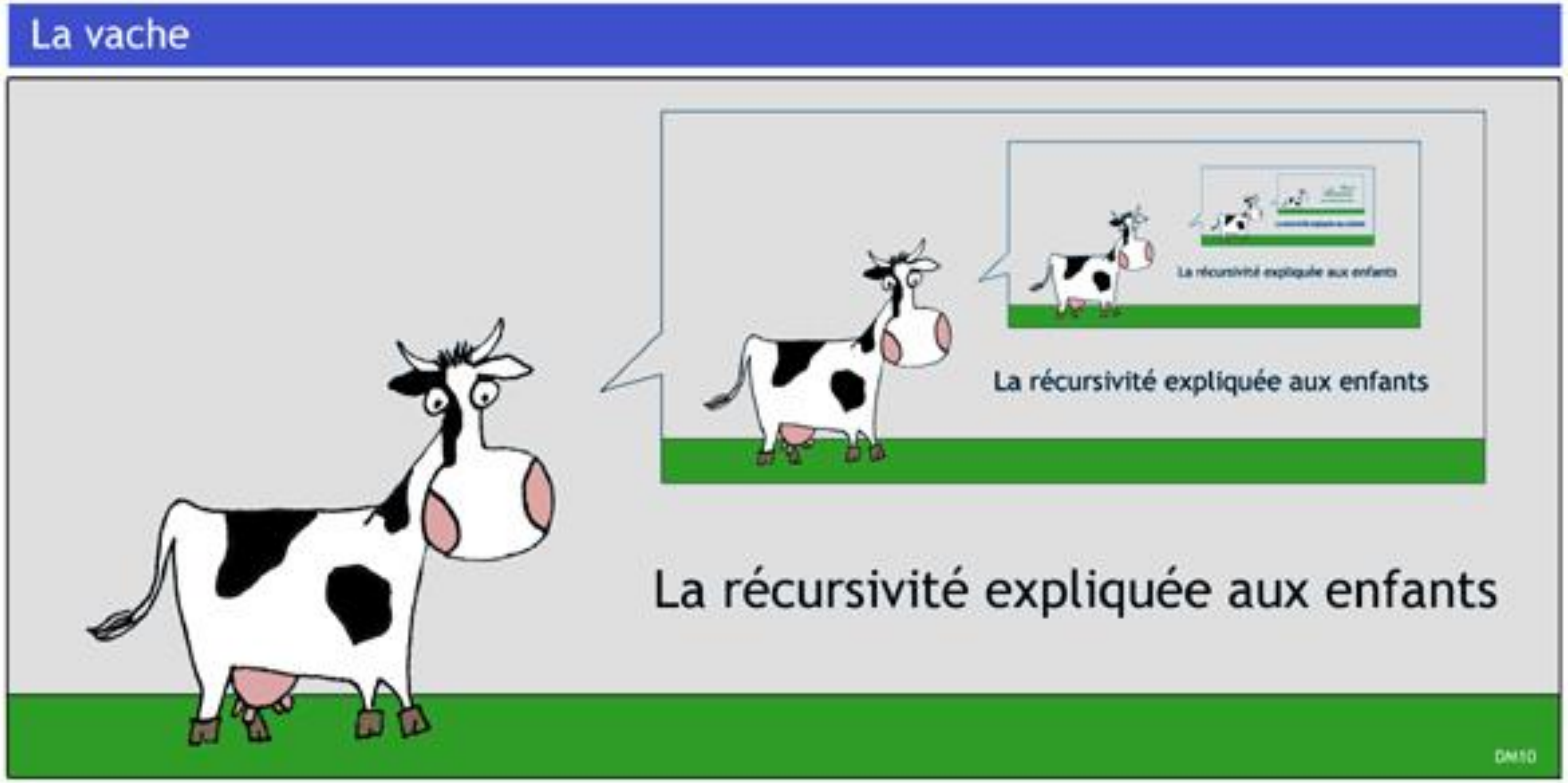
# ALGORITHMIQUE AVANCEE
# Chapitre 1: La récursivité

Semestre 3
Licence 2 Info.



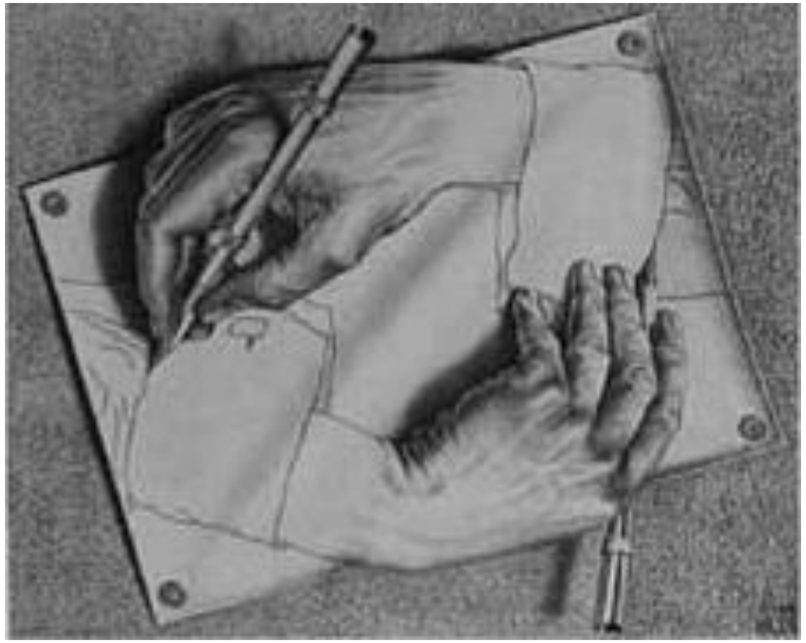
# Sommaire :

# 1. Introduction

## Définitions

- Une construction est dite **récursive,** si elle se définit à partir d’elle-même.

- On qualifie de **récursive,** toute **fonction ou procédure** qui s’appelle elle même.



- **Qu'est-ce que la programmation récursive ?**

  la programmation récursive est une technique de programmation qui remplace les instructions de boucle (**while**, **for** et **do while**.) par des appels de fonction.

# 1. Introduction

## Exemple 1:

Ecrire une fonction qui affiche le message « *Bonjour…*» dix fois à l'écran sans utiliser de boucle ni répéter plusieurs fois les *cout*.

```
#include<iostream>
using namespace std;

void foncRec_1()
{
    cout << "Bonjour..." << endl;
    foncRec_1(); // Appel récursif de la procédure
}

int main()
{
    foncRec_1();  //Appel de la procédure
    return 0;
}
```

**Remarque**: le message « *Bonjour…* » s'affiche en boucle infinie.

**Note : L'appel récursif est traité comme n'importe quel appel de fonction.**

# 2.Comment écrire un algorithme récursif ?

Pour écrire un algorithme récursif il faut tout d’abord analyser le problème à résoudre pour pouvoir identifier :

- **le ou les cas particuliers (aussi dit de bases);**
- **le cas général qui effectue la récursion.**

**Cas particulier :**

on décrit les cas pour lesquels le résultat de la fonction est simple à calculer : la valeur retournée par la fonction est directement définie.

**Cas général :**

la fonction est appelée récursivement et le résultat retourné est calculé en utilisant le résultat de l'appel récursif. **A chaque appel récursif, la valeur d'au moins un des paramètres (effectifs) de la fonction doit changer.**

**Attention !**

**Il faut toujours s'assurer que chaque cas général converge vers un cas de base.**

# 2.Comment écrire un algorithme récursif ?

## Cas de bases/Cas général :

**Factorielle**

**Cas de base :** $0! = 1! = 1$

**Cas général :** $n! = n(n-1)!$

**Suite de fibonacci**

**Cas de base :**
$F(0) = 0$
$F(1) = 1$

**Cas général :** $F(n) = F(n-1) + F(n-2), n > 1$

**Somme des n premiers entiers**

**Cas général :** $\sum_{i=1}^{n} i = n + \sum_{j=1}^{n-1} j$

**Cas de base :** $\sum_{i=1}^{n} i = 1, \text{ pour } n = 1$

# 2.Comment écrire un algorithme récursif ?

## Solution exemple 1:

```
#include<iostream>
using namespace std;

void foncRec_2(int i)
{
    if (i == 10)  // Condition d'arrêt
    {
        return;
    }
    cout << "Bonjour..." << endl;
    foncRec_2(i + 1); /*Appel récursif de la procédure
                        avec évolution du paramètre*/
}

int main()
{
    foncRec_2(0);  //Appel de la procédure
    return 0;
}
```

- ❑ Puisqu'une fonction récursive s'appelle elle-même, il est impératif de prévoir une **condition d'arrêt** à la récursivité, sinon le **programme ne s'arrête jamais**

- ❑ On doit toujours **tester en premier la condition d'arrêt,** et ensuite, si la condition n'est pas vérifiée, lancer un **appel récursif**.

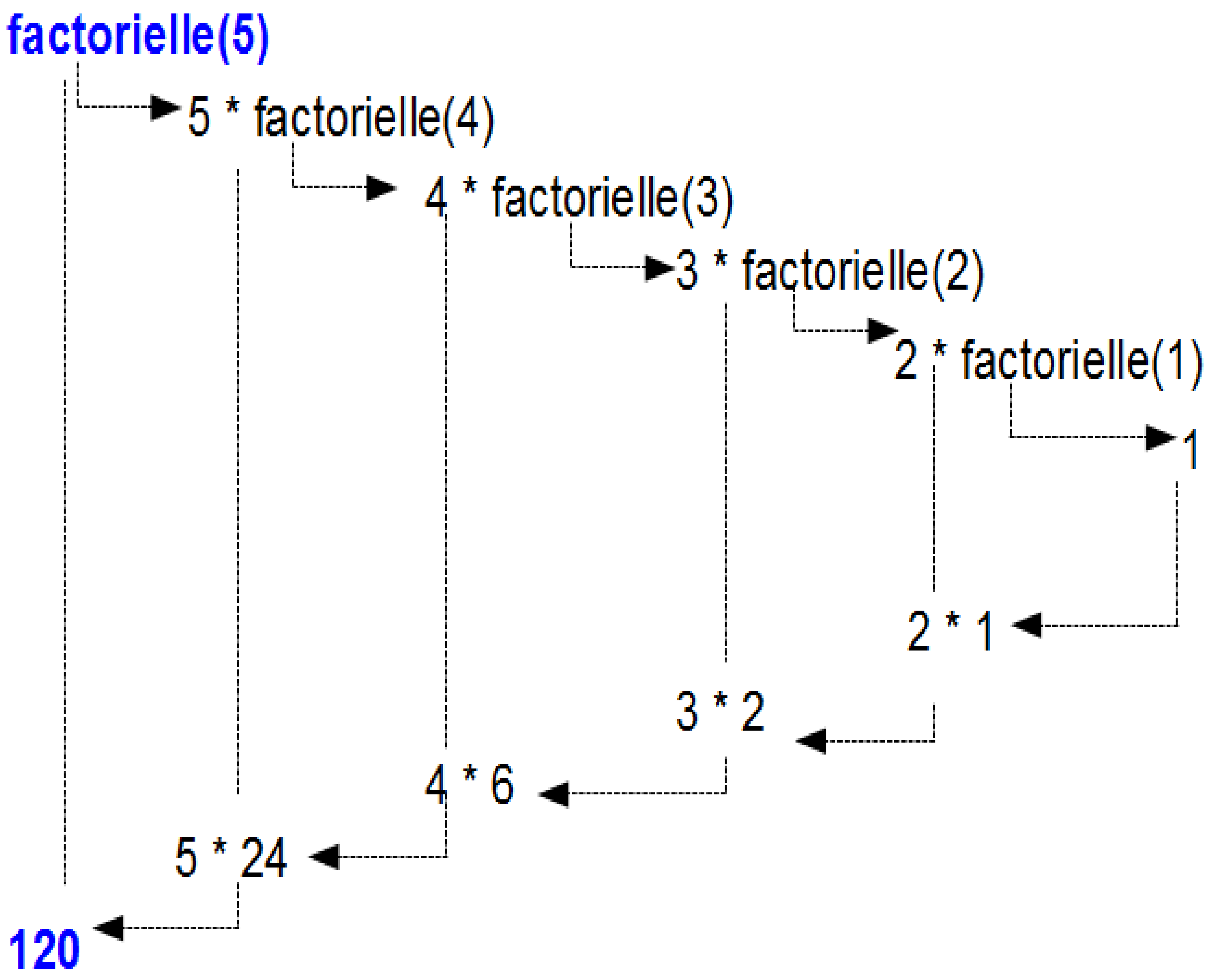
# 2.Comment écrire un algorithme récursif ?

**Exemple 2 :**

**Factorielle**

```
fonction fact (n : Naturel) : Naturel
debut
    si n=0 ou n=1 alors
        retourner 1
    sinon
        retourner n*fact(n-1)
    finsi
fin
```

**Comment ça marche ?**



# 2.Comment écrire un algorithme récursif ?

## Récursivité terminale

**Définition**

L'appel récursif est la dernière instruction et elle est isolée

**plus(a,b)**

```
fonction plus (a,b : Naturel) : natuel
debut
    si b=0 alors
        retourner a
    sinon
        retourner plus(a+1,b-1)
    finsi
fin
```

plus(4,2)=plus(5,1)=plus(6,0)=6

# 2.Comment écrire un algorithme récursif ?

## Récursivité non terminale

**Définition**

L'appel récursif n'est pas la dernière instruction et/ou elle n'est pas isolée (fait partie d'une expression)

**plus(a,b)**

```
fonction plus (a,b : Naturel) : natuel
debut
    si b=0 alors
        retourner a
    sinon
        retourner 1+plus(a,b-1)
    finsi
fin
```

plus(4,2)=1+plus(4,1)=1+1+plus(4,0)=1+1+4=6

# 3. Dérécursivation

Il est possible de transformer de façon simple une fonction récursive terminale en une fonction itérative : c'est la **dérécursivation**.

- Une fonction **récursive terminale** a pour forme générale :

```
fonction avec retour T recursive(P)
début
      I0
      si  (C) alors
           I1
      sinon
           I2
           recursive(f(P));
      finsi
fin
```

**T** est le type de retour
**P** est la liste de paramètres
**C** est la condition d'arrêt
**I0** le bloc d'instructions exécuté dans tous les cas
**I1** le bloc d'instructions exécuté si C est vraie
**I2** et le bloc d'instructions exécuté si C est fausse
**f** la fonction de transformation des paramètres

La **fonction itérative** correspondante est :

```
fonction avec retour T iterative(P)
début
      I0
      tantque (non C) faire
           I2
           P <- f(P);
           I0;
      fintantque
      I1
fin
```

# 3. Dérécursivation

## Exemple : Dérécursivation de la factorielle terminale

```
#include <iostream>

using namespace std;


int factorielle_Rec(int n, int fact)
{
    if((n==0)||(n==1)) return fact;
    else return factorielle_Rec(n-1,n*fact);
}


int factorielle_Iter(int n, int fact)
{
    while(n>1){
        fact = n * fact;
        n--;
    }
    return fact;
}

int main()
{
    // les deux fonction doivent être appelé en mettant le deuxième
    // parametre à 1     fact= 1

    cout<<factorielle_Iter(3,1)<<endl;
    cout<<factorielle_Rec(3,1)<<endl;
    return 0;
}
```

# 3. Dérécursivation

- Une fonction **récursive terminale** a pour forme générale :

```
fonction avec retour T recursive(P)
début
      I0
      si (C) alors
           I1
      sinon
           I2
           recursive(f(P));
      finsi
fin
```

**T** est le type de retour
**P** est la liste de paramètres
**C** est la condition d'arrêt
**I0** le bloc d'instructions exécuté dans tous les cas
**I1** le bloc d'instructions exécuté si C est vraie
**I2** et le bloc d'instructions exécuté si C est fausse
**f** la fonction de transformation des paramètres

La **fonction itérative** correspondante est :

```
fonction avec retour T iterative(P)
début
      I0
      tantque (non C) faire
           I2
           P <- f(P);
           I0;
      fintantque
      I1
fin
```

# 4.Utiles à savoir

## Les boucles `for`

- Très bonne candidate
- Toute boucle `for` peut se transformer en une fonction récursive
- Principe :
  - Pour faire des choses pour un indice allant de 1 à *n*
    - On les fait de 1 à *n*-1 (même traitement avec une donnée différente)
    - Puis on les fait pour l'indice *n* (cas particulier)

## Traduction

```
void f(int n)  {
  for (int i=0; i<=n;
  i++)
     traiter(i);
}
```

```
void f(int n){
  if(n==0)
     traiter(0);
  else {
     f(n-1);
     traiter(n);
  }
}
```

# Exemple : fonction factorielle

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

```
int fact(int n){
  if(n==0)
    return 1;
  else
    return
    fact(n-1)*n;
}
```

# 4.Utiles à savoir

## Quelques conséquences

- La plupart des traitement sur les tableaux peuvent se mettre sous forme récursive :
  - Tris (sélection, insertion)
  - Recherche séquentielle (attention: pas dichotomique)
  - Inversion
  - Problème des huit reines
  - Etc...

## Une constatation

- L'écriture sous forme récursive est toujours plus simple que l'écriture sous forme itérative

# 4.Utiles à savoir

## Une question

- Une même fonction est-elle plus efficace sous forme récursive ou sous forme itérative ? (Ou, sous une autre forme, y a-t-il un choix optimal généralisable ?)
- La réponse est non. La réponse à la question inverse est non. Il n'y a pas de généralité

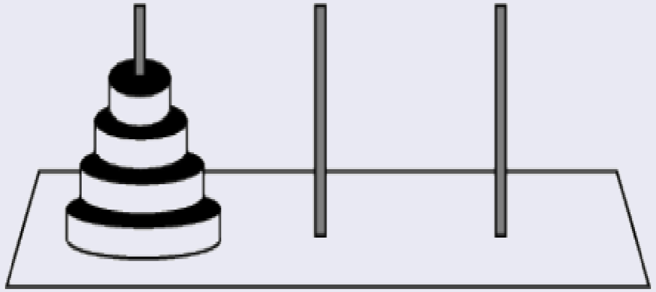
## En revanche

- La plupart des traitements itératifs simples sont facilement traduisibles sous forme récursive (exemple du `for`)
- L'inverse est faux
- Il arrive même qu'un problème ait une solution récursive triviale alors qu'il est très difficile d'en trouver une solution itérative

# 5. Exemple pratique

## Les tours de Hanoï 1 / 4

### Présentation



Les tours de hanoï est un jeu solitaire dont l'objectif est de déplacer les disques qui se trouvent sur une tour (par exemple ici la première tour, celle la plus à gauche) vers une autre tour (par exemple la dernière, celle la plus à droite) en suivant les règles suivantes :

- on ne peut déplacer que le disque se trouvant au sommet d'une tour ;
- on ne peut déplacer qu'un seul disque à la fois ;
- un disque ne peut pas être posé sur un disque plus petit.

## Les tours de Hanoï 2 / 4

### Opérations disponibles

**procédure** dépilerTour (**E/S** t : TourDeHanoi,**S** d : Disque)
**procédure** empilerTour (**E/S** t : TourDeHanoi,**E** d : Disque)

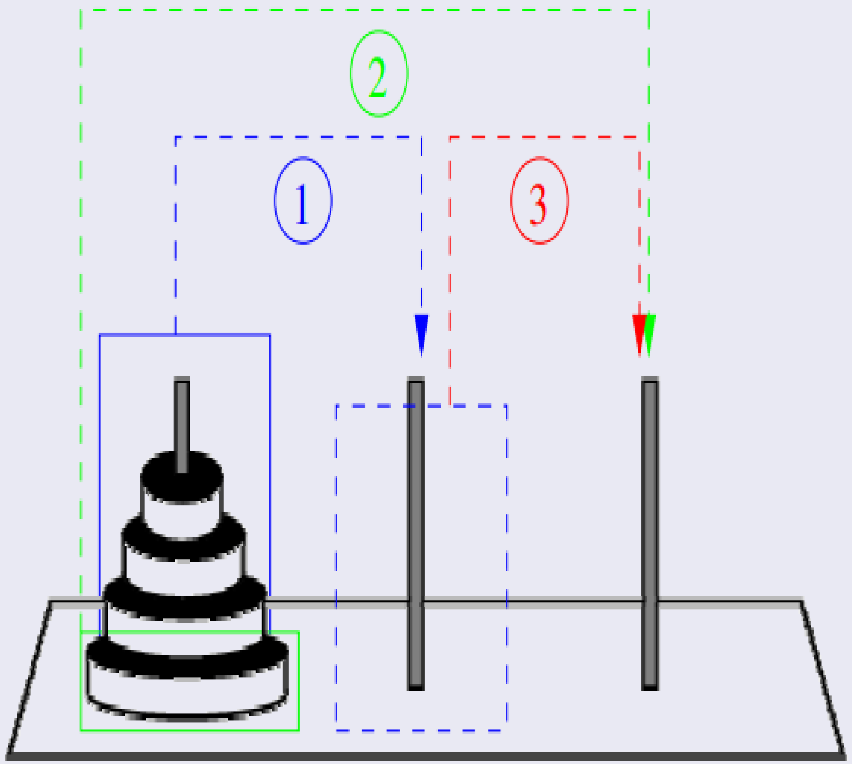
### Objectif

**procédure** resoudreToursDeHanoi (**E** nbDisquesADeplacer : **Naturel**, **E/S** source, destination, intermediaire : TourDeHanoi)

# 5. Exemple pratique

## Les tours de Hanoï 3 / 4

**Analyse du problème**




## Les tours de Hanoï 4 / 4

**Solution**

**procédure** resoudreToursDeHanoi (**E** nbDisquesADeplacer : **Naturel**, **E/S** source, destination, intermediaire : TourDeHanoi)

    **Déclaration** d : Disque

**debut**

    **si** nbDisquesADeplacer>0 **alors**

        resoudreToursDeHanoi(nbDisquesADeplacer-1, source, intermediaire, destination)

        depiler(source,d)

        empiler(destination,d)

        resoudreToursDeHanoi(nbDisquesADeplacer-1, intermediaire, destination, source)

    **finsi**

**fin**

# Exercice

## Exercice

Écrire le programme qui calcule le plus grand commun dénominateur (pgcd) de deux entiers $a$ et $b$.

<u>Méthode des différences :</u>
Si $a$ et $b$ sont multiples de $d$ alors $a - b$ également.
Réciproquement, si $d$ divise $b$ et $a - b$ alors il divise également $(a - b) + b = a$.

<u>Calcul du pgcd :</u>
pgcd(a, 0) = a
pgcd(0, b) = b
pgcd(a, b)= pgcd(a, b-a) si a < b
pgcd(a, b) = pgcd(a-b, b) sinon

## Solution

pgcd(m,n) :

- 2 cas de base :
  - Si a == 0 alors pgcd(b,a) = b
  - Si b == 0 alors pgcd(b,a) = a
- 2 cas généraux :
  - Si b < a alors pgcd(b, a)= pgcd(b, a-b)
  - Sinon pgcd(b, a) = pgcd(b-a,a)

```
int pgcd (int b, int a){
    if(a == 0) return b;
    if(b == 0) return a;

    if(b < a)
      return pgcd(b, a-b);
    return pgcd(b-a, a);
}
```

# 6. Conclusion

**En conclusion**

- Les algorithmes récursifs sont simples (c'est simplement une autre façon de penser)
- Les algortihmes récursifs permettent de résoudre des problèmes complexes
- Il existe deux types de récursivités :
  - terminale, qui algorithmiquement peuvent être transformée en algorithme non récursif
  - non terminale
- Les algorithmes récursifs sont le plus souvent plus gourmands en ressource que leurs équivalents itératifs