# Récursivité et Paradigmes algorithmiques

## Recursivité

- Une fonction $f$ est **récursive** si elle s'appelle elle-même. Elle est composée de :
  - Un (ou plusieurs) **cas de base** où $f$ renvoie directement une valeur, sans appel récursif.
  - Un (ou plusieurs) **appel récursif** à $f$ avec des arguments plus petits que ceux de l'appel initial, qui garantit de se ramener au cas de base.

  Exemple : Calcul de $n! = n \times (n-1)!$.

```python
def fact(n):
    if n == 0: # cas de base
        return 1
    return n*fact(n - 1) # appel récursif
```

- La fonction suivante calcule les termes de la suite de Fibonacci ($u_0 = u_1 = 1, u_n = u_{n-1} + u_{n-2}$) :

```python
def fibo(n):
    if n == 0 or n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

  Cependant, la complexité est très mauvaise (exponentielle)... En effet, si $C(n)$ = complexité de `fibo(n)`, alors $C(n) = \underbrace{C(n-1)}_{fibo(n-1)} + \underbrace{C(n-2)}_{fibo(n-2)} + K$ ce qui est une équation récurrente linéaire d'ordre 2 (du même type que celle vérifiée par la suite de Fibonacci) que l'on peut résoudre pour trouver $C(n) = \Theta(\varphi^n)$ où $\varphi$ est le nombre d'or.
  Le soucis vient du fait que le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.

## Diviser pour régner

Cette approche consiste à résoudre des sous-problèmes indépendants (majoritairement récursivement) puis combiner les sous-résultats pour obtenir une solution du problème initial.

## Programmation dynamique

- La **programmation dynamique** stocke les résultats des sous-problèmes non-indépendants afin d'éviter de les calculer plusieurs fois.

```python
def fibo(n):
    L = [1, 1] # L[i] = ième terme de Fibonacci
    for i in range(n - 1):
        L.append(L[i] + L[i + 1]) # récurrence
    return L[n]
```

- Pour résoudre un problème de programmation dynamique :
  1. Caractériser la structure d'une solution optimale
  2. Chercher une équation de récurrence entre sous-solutions optimales et déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.
  3. Calculer la valeur d'une solution optimale au problème, avec une approche bottom-up ou top-down. Il faut stocker en mémoire (dans une collection appropriée) les résultats des sous-problèmes dépendants pour éviter de les calculer plusieurs fois.

- Une approche **bottom-up** consiste à partir du (ou des) cas de base et de calculer au travers d'une boucle les solutions aux sous-problèmes.

  Exemple d'approche bottom-up : Calcul de $\binom{n}{k}$ en utilisant la formule de Pascal $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ et les cas de base $\binom{n}{0} = 1$ et $\binom{n}{0} = 0$ si $n \neq 0$.

  On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

```python
def binom(n, k):
    M = [[0]*(k + 1) for _ in range(n + 1)]
    for i in range(0, n + 1):
        M[i][0] = 1
    for i in range(1, n + 1):
        for j in range(1, k + 1):
            M[i][j] = M[i - 1][j - 1] + M[i - 1][j]
    return M[n][k]
```

- La **mémoïsation** est l'approche **top-down** de la programmation dynamique.
  Pour éviter de résoudre plusieurs fois le même sous-problème, on mémorise (dans une structure appropriée) les arguments pour lesquelles la fonction récursive a déjà été calculée. Il est impératif que la fonction soit une **fonction pure**, c'est-à-dire, qu'elle verifie les deux propriétés suivantes :
  - Sa valeur de retour est invariante pour les mêmes arguments.
  - Son évaluation ne provoque pas d'effets de bord.

- Version mémoïsée du calcul de la suite de Fibonacci :

```python
def fibo(n):
    d = {} # d[k] contiendra le kème terme de la suite
    def aux(k):
        if k == 0 or k == 1:
            return 1
        if k not in d:
            d[k] = aux(k - 1) + aux(k - 2)
        return d[k]
    return aux(n)
```

- Mémoïsation du calcul de coefficient binomial :

```python
def binom(n, k):
    d = {}
    def aux(i, j):
        if j == 0: return 1
        if i == 0: return 0
        if (i, j) not in d:
            d[(i, j)] = aux(i - 1, j - 1) + aux(i - 1, j)
        return d[(i, j)]
    return aux(n, k)
```