- 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 $n! = n \times (n-1)!$.

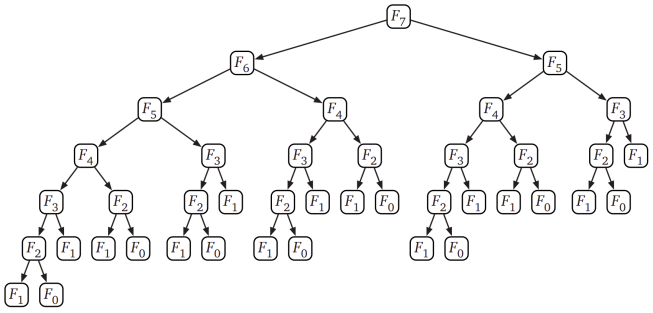
```python
def fact(n):
    if n == 0: # cas de base
        return 1
    return n*fact(n - 1) # appel récursif
```

  Attention : une fonction récursive peut ne pas terminer (appels récursifs infinis), de même qu'un `while` peut faire boucle infinie.

- 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 $C(n) \sim Ar^n$.
  Le soucis vient du fait que le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.



Appels récursifs de la version naïve de `fibo(7)`

- La **programmation dynamique** stocke les résultats des sous-problèmes pour 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. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
  2. Déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.
  3. Stocker en mémoire (dans une liste, matrice ou dictionnaire) les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

- Exemple : 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 similaire à la programmation dynamique mais en utilisant une fonction récursive plutôt que des boucles.
  Pour éviter de résoudre plusieurs fois le même problème (comme pour Fibonacci), on mémorise (dans un tableau ou un dictionnaire) les arguments pour lesquelles la fonction récursive a déjà été calculée.

- 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)
```

- Problème du sac à dos.
  Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n. On suppose que les poids sont strictement positifs.
  Sortie : la valeur maximum que l'on peut mettre dans le sac.
  Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

$$dp[i][0] = 0$$

$$dp[i][j] = \max(\underbrace{dp[i][j-1]}_{\text{sans prendre } o_j}, \underbrace{dp[i-w_j][j-1]+v_j}_{\text{en prenant } o_j, \text{si } i-w_j \geq 0})$$

Résolution par programmation dynamique :

```python
def knapsack(c, w, v):
    """
    Renvoie la valeur maximum que l'on peut mettre
    dans un sac à dos de capacité c.
    Le ième objet a pour poids w[i] et valeur v[i].
    """
    n = len(w) # nombre d'objets
    dp = [[0]*(n + 1) for i in range(c + 1)]
    # dp[i][j] = valeur max dans un sac de capacité i
    # où j est le nombre d'objets autorisés
    for i in range(1, c + 1):
        for j in range(1, n + 1):
            if w[j - 1] <= i:
                x = v[j - 1] + dp[i - w[j - 1]][j - 1]
                dp[i][j] = max(dp[i][j - 1], x)
            else:
                dp[i][j] = dp[i][j - 1]
    return dp[c][n]
```

Résolution par mémoïsation :

```python
def knapsack_memo(c, w, v):
    dp = {}
    def aux(i, j):
        if i == 0 or j == 0:
            return 0
        if (i, j) not in dp:
            dp[(i, j)] = aux(i, j - 1)
            if w[j - 1] <= i:
                x = v[j - 1] + aux(i - w[j - 1], j - 1)
                dp[(i, j)] = max(dp[(i, j)], x)
        return dp[(i, j)]
    return aux(c, len(w))
```