# PROGRAMMATION DYNAMIQUE

Informatique Tronc Commun

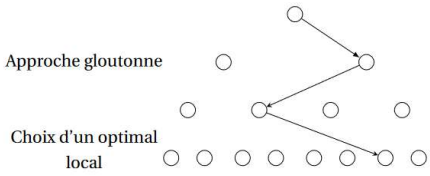
E. CLERMONT

# INTRODUCTION

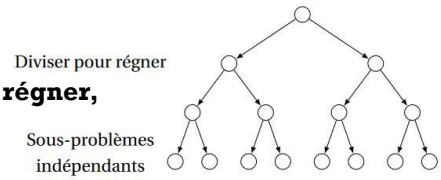
**Algorithmes de décomposition** : algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre.

**3 familles** :

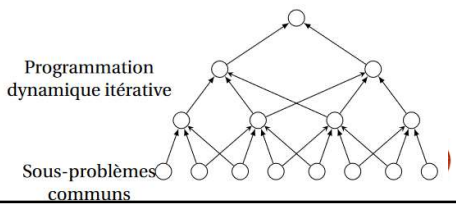
- les algorithmes **gloutons,**




- les algorithmes de type **diviser pour régner,**



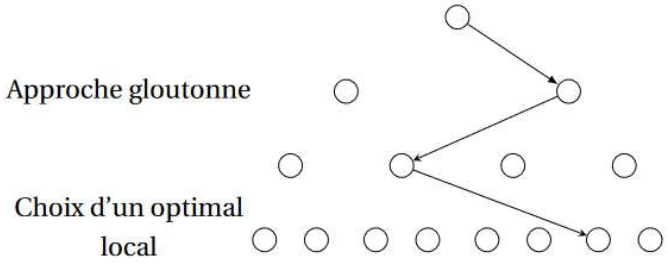

- la **programmation dynamique**.



# INTRODUCTION

**Algorithmes de décomposition** : algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre

- **3 familles** :
  - les algorithmes **gloutons,**
  - les algorithmes de type **diviser pour régner,**
  - la **programmation dynamique**.




- **Limites** des algorithmes de type gloutons

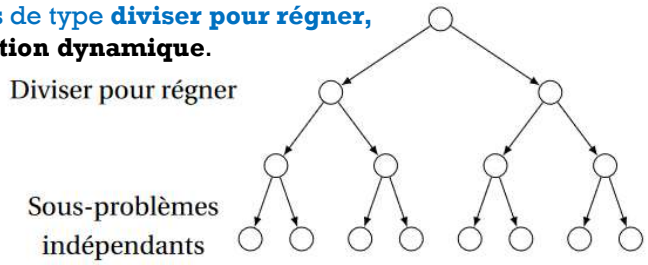
Même s'il existe des algorithmes gloutons optimaux (càd,qui produisent une solution optimale au problème), la plupart du temps ce n'est pas le cas.

3

# INTRODUCTION

**Algorithmes de décomposition** : algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre

- **3 familles** :
  - les algorithmes **gloutons,**
  - les algorithmes de type **diviser pour régner,**
  - la **programmation dynamique**.




- **Limites** des algorithmes de type diviser pour régner

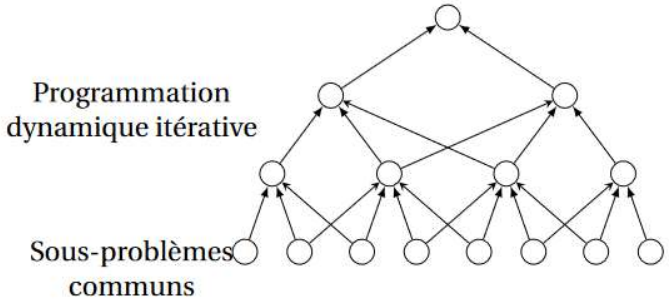
L'approche **diviser pour régner** est très efficace pour de nombreux problèmes , mais elle nécessite que les **sous-problèmes soient indépendants**. Or, parfois, il n'en est rien, certains sous-problèmes ont des **sous-problèmes en commun**, ils ne sont pas indépendants, ils se chevauchent. Dans ce cas, l'approche diviser pour régner devient inefficace puisqu'elle résout plusieurs fois les mêmes sous-problèmes.

4

# INTRODUCTION

**Algorithmes de décomposition** : algorithmes qui cherchent à décomposer un problème en sous-problèmes afin de le résoudre

- **3 familles** :
  - les algorithmes **gloutons,**
  - les algorithmes de type **diviser pour régner,**
  - la **programmation dynamique**.




- Pour dépasser ces limites, on étudie la **programmation dynamique**.

5

# PRINCIPES

- La programmation dynamique (*dynamic programing* ou encore *dynamic optimization* en anglais) est un paradigme de programmation, c'est-à-dire une façon particulière d'appréhender un problème algorithmique donné.

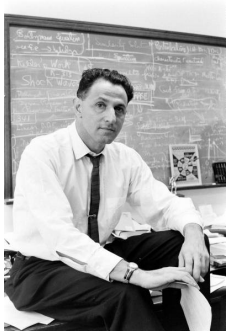
- Le terme « programmation » ne correspond pas à l'utilisation d'un langage de programmation donné, mais à une structure de raisonnement planifiant et ordonnançant les données nécessaires à la résolution de problèmes.

- C'est une méthode utile pour obtenir une solution exacte à un problème algorithmique, là où une solution « classique » est trop complexe, c'est-à-dire trop peu efficace.

On parle alors d'**optimisation combinatoire**.

6

# PRINCIPES



- Ce concept a été introduit par Richard Bellman, dans les années 1950, pour résoudre typiquement des problèmes d'optimisation

- Cette technique de programmation suit le **principe d'optimalité de Bellman** :
- la solution optimale d'un problème peut être calculée à partir de solutions optimales de sous-problèmes.

*« Toute politique optimale est composée de sous-politiques optimales ».*

7

# PRINCIPES

- La programmation dynamique

=> méthode de construction des solutions optimales d'un problème par combinaison des solutions optimales de sous-problèmes.

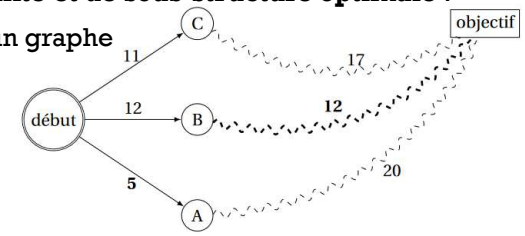
- Certaines combinaisons sont implicitement rejetées si elles appartiennent à un sous-ensemble qui n'est pas intéressant : on ne construit que les solutions optimales **des sous-problèmes utiles** à la construction de la solution optimale.
- Cette approche :
  • **décompose** le problème P **en sous-problèmes** de taille moindre,
  • ne résout un **sous-problème qu'une seule fois**.
- En général, le cadre de l'application de la programmation dynamique sont les problèmes d'optimisation combinatoire dont la sous-structure est optimale. Pour ce genre de problèmes, il y a de nombreuses solutions possibles.
- Chaque solution possède une valeur propre que l'on peut quantifier.
- On cherche alors soit à la minimiser soit à la maximiser, dans tous les cas, on cherche au moins **une valeur optimale**.

8

# PRINCIPES

- Illustration du **principe d'optimalité et de sous-structure optimale** :

trouver le plus court chemin dans un graphe




- Résoudre ce problème via la programmation dynamique suppose qu'on a un problème à sous-structure optimale, ce qui est le cas.
- On peut exprimer le plus court chemin récursivement en fonction du 1er chemin choisi et du reste du graphe.
- Cela peut s'écrire formellement mathématiquement ou, plus simplement, ainsi:
  **Le plus court chemin** est le chemin le plus court **à choisir** parmi :
  • le chemin qui passe par A suivi par le chemin le plus court de A à l'objectif,
  • le chemin qui passe par B suivi par le chemin le plus court de B à l'objectif,
  • le chemin qui passe par C suivi par le chemin le plus court de C à l'objectif.
  La résolution du problème amènera à choisir le chemin qui passe par B.

9

# PRINCIPES

- **Notion de sous-structure optimale** :

la solution optimale à un problème de taille n (ayant n éléments) est basée sur une solution optimale au même problème de plus petite taille (moins de n éléments).

- c'est-à-dire que, tout en construisant la solution d'un problème de taille n, on la définit en fonction de problèmes similaires de taille plus petite. On trouve les solutions optimales de moins d'éléments et on combine les solutions pour obtenir le résultat final.

- Un problème présente **une sous-structure optimale** si une solution optimale peut être construire **à partir des solutions optimales à ses sous-problèmes.**

10

## PRINCIPES

2 conditions pour la programmation dynamique :

- la sous-structure optimale
- le **chevauchement des sous-problèmes**.

- La programmation dynamique désigne une classe d'algorithmes qui sont **généralement de complexité polynomiale ($O(n^k)$ ).**
- Contrairement aux algorithmes gloutons, ils permettent de traiter des problèmes dont la solution naïve est exponentielle $O(2^n)$ .
- C'est une technique d'optimisation d'un algorithme visant à éviter de recalculer des sous-problèmes en stockant les résultats en mémoire.
- L'idée est simple, mais le gain sur la complexité en temps peut être considérable, et cette technique est très largement utilisée dans de nombreux algorithmes.
- Une autre caractéristique est qu'ils utilisent plus de mémoire. Pour gagner en efficacité, on "échange de la mémoire contre du temps"

11

## EXEMPLE INTRODUCTIF: LA SUITE DE FIBONACCI

**Exemple classique de la suite de Fibonacci:**

- La suite de Fibonacci est la suite d'entiers $(u_n)$ avec $n \geq 0$ définie récursivement par :

**Code partage : 6ac4-2403581**

$u_0 = 0$
$u_1 = 1$
$un = u_{n-1} + u_{n-2} \quad \forall n \geq 2$

- Approche « naïve »

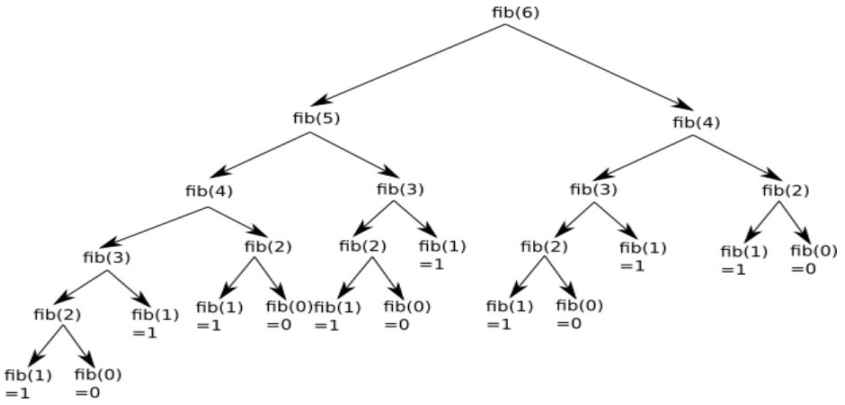
```
def fib(n):
    """Version récursive naïve"""
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)

print ("version naive",fib(10))
print ("version naive",fib(15))
```

Cette solution est récursive de **complexité $O(2^n)$** : complexité exponentielle donc inefficace algorithmiquement.

12

## EXEMPLE INTRODUCTIF: LA SUITE DE FIBONACCI

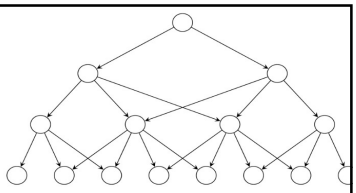
- **Raison de cette inefficacité** : multiplicité de calculs d'un même nombre.




Une solution pour gagner en efficacité : **éviter la multiplicité de résolutions** du même sous-problème.
On améliore beaucoup la complexité temporelle si, une fois calculé, on sauvegarde un résultat, par exemple dans une liste ou un dictionnaire. Quand on en a besoin, on récupère la valeur calculée dans cette liste.

13

## APPROCHE RÉCURSIVE AVEC MÉMOÏSATION (TOP DOWN)



C'est une approche récursive pour laquelle le principe est :

- d'utiliser **une formule de récurrence**;
- avant tout calcul, de regarder **si le résultat est déjà stocké** dans une liste (ou dans un dictionnaire).

Intuitivement, on comprend que la complexité d'une telle approche est largement meilleure que la solution récursive initiale car il y a beaucoup moins de calculs.

Le terme "Top Down" prend alors tout son sens : on part du haut ("top") du problème (le problème initial) et on le découpe pour arriver à résoudre des problèmes de tailles plus petites ("down").

14

## APPROCHE RÉCURSIVE AVEC MÉMOÏSATION (TOP DOWN)

Pour la suite de Fibonacci (avec stockage **dans une liste**) :

```
def fibonacci_memoisation(n , L):
  """Version récursive mémoïsation (Top Down)"""
  if n <= 1 :
     return n
  elif L[n] > 0:
     return L[n]
  else:
     L[n] = fibonacci_memoisation(n-1,L)
               + fibonacci_memoisation(n-2 , L)
    return L[n]
#test
n = 10
L = [0] * (n+1)
res =  fibonacci_memoisation(n , L)
print( "fibonacci memoisation : ", res )

# ou avec une fonction :
def fib(n):
    L = [0] * (n+1)
    return  fibonacci_memoisation(n , L)
print ("mémoïsation (Top Down)",fib(10))
```

Complexité: O(n)

## APPROCHE RÉCURSIVE AVEC MÉMOÏSATION (TOP DOWN)

Pour la suite de Fibonacci (avec stockage **dans un dictionnaire)** :

```
# approche Programmation dynamique mémoisation avec dictionnaire
# Complexite :  O(n)
def fibonacci_memoisation_dictionnaire(n , dico ):
  """Version récursive mémoïsation dictionnaire(Top Down)"""
  if n <= 1:
     return n
  elif n in dico:
     return dico[n]
  else:
     dico[n] = fibonacci_memoisation_dictionnaire(n-1 , dico)
             + fibonacci_memoisation_dictionnaire(n-2 , dico)
     return dico[n]
def fib(n):
    dico = {}
    return  fibonacci_memoisation_dictionnaire(n , dico)
print ("mémoïsation dictionnaire (Top Down)",fib(10))
```

## APPROCHE RÉCURSIVE AVEC MÉMOÏSATION (TOP DOWN)

En Python, on peut réaliser la mémoïsation à l'aide de décorateurs.

On écrit une fonction *memoize* qui prend en argument une fonction (récursive) et qui doit la modifier pour appliquer la mémoïsation.

Ensuite, on utilise un **décorateur *@memoize*** avant la définition de la fonction *fib*.

```
# Approche memoisation avec annotation
def memoize(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
print (fib(10))
```

17

## APPROCHE RÉCURSIVE AVEC MÉMOÏSATION (TOP DOWN)

Il est également possible d'utiliser directement le décorateur *cache LRU* (pour *Least Recently Used*) de la bibliothèque functools de Python.

Il est alors intéressant de pouvoir avoir des statistiques sur l'utilisation du cache (*cache_info*).
https://docs.python.org/3/library/functools.html

```
from functools import lru_cache
@lru_cache
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
print (fib(10))

[fib(n) for n in range(16)]
print( fib.cache_info() )
```

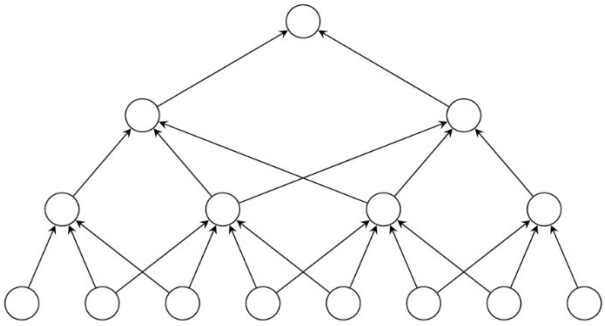
18

# MÉTHODE ASCENDANTE BOTTOM UP (OU DOWN TOP)

C'est une **approche itérative** pour laquelle :

- on commence par résoudre les sous-problèmes de la taille la plus petite à ceux de la taille la plus grande,
- on stocke les résultats au fil de la résolution des sous-problèmes.

Le terme "Bottom Up" : on part du plus petit problème ("bottom") pour aller au plus grand ("up").



19

# MÉTHODE ASCENDANTE BOTTOM UP (OU DOWN TOP)

Stockage **dans une liste**

```
def fibonacci_BottomUp(n): # avec une liste
    """Version ITERATIVE Bottom Up avec liste"""
    L = [0] * (n+1)
    L[1] = 1
    for i in range(2 , n+1):
        L[i] = L[i-1] + L[i-2]
    return L[n]
print ("Bottom Up",fibonacci_BottomUp(10))
```

Complexité: O(n)

```
def fibonacci_BottomUpv2(n): # en ne conservant que les 2
    derniers termes => meilleure complexité mémoire
    """Version ITERATIVE Bottom Up avec liste"""
    L = [0] * (2)
    L[1] = 1
    for i in range(2 , n+1):
        tmp = L[0]
        L[0] = L[1]
        L[1] = L[0] + tmp
    return L[1]
print ("Bottom Up v2",fibonacci_BottomUpv2(10))
```

## MÉTHODE ASCENDANTE BOTTOM UP (OU DOWN TOP)

Pour la suite de Fibonacci (avec stockage **dans un dictionnaire)**

```
def fibonacci_BottomUp_dictionnaire(n):
    """Version ITERATIVE Bottom Up avec dictionnaire"""
    dico = {}
    dico[0] = 0
    dico[1] = 1
    for i in range(2 , n+1):
        dico[i] = dico[i-1] + dico[i-2]
    return dico[n]
print("Bottom Up dictionnaire",fibonacci_BottomUp_dictionnaire(10))
```

21

## EXEMPLE INTRODUCTIF: LA SUITE DE FIBONACCI

- Remarque 1 : Une autre version de cette fonction existe n'utilisant pas le principe de la programmation dynamique et qui est encore meilleure (temps toujours en O(n) mais espace mémoire en O(1)).
- Possible car le principe d'optimalité des termes de la suite est très simple (dépendance directe avec les deux termes précédents).

```
# approche en O(n) sans programmation dynamique
def fibo (n):
    n1 = 0
    n2 = 1
    for k in range(n):
        temp = n1 + n2
        n1 = n2
        n2 = temp
    return n1
print (fibo(10))
```

22

## EXEMPLE INTRODUCTIF: LA SUITE DE FIBONACCI

- Remarque 2 : en analysant la structure des appels récursifs, une autre version récursive qui se base sur le principe qu'il n'est pas nécessaire de tout mémoriser.
- Cette version ne fait aussi qu'un seul appel récursif mais cette fois, n'utilise pas un tableau pour stocker toutes les valeurs intermédiaires mais fait juste remonter les deux valeurs précédentes.

```
def fibonacci2 (n, last = True):
    if n == 0 :
        res = (0,0)
    elif n == 1 :
        res = (0,1)
    else :
        (n1, n2) = fibonacci2(n-1,False)
        res = (n2 , n1 + n2)
    if last :
        return res[1]
    else :
        return res
print (fibonacci2(10))
```

23

## PROGRAMMATION DYNAMIQUE : UN COMPROMIS ENTRE COMPLEXITÉ TEMPORELLE ET EN MÉMOIRE

- **Optimisation**
- La programmation dynamique est l'exemple parfait de **compromis entre complexité en temps et complexité en mémoire.**
- En stockant les résultats, et donc en augmentant la complexité en mémoire, on arrive à réduire radicalement la complexité en temps.
- Dans notre cas, l'optimisation est très intéressante car elle permet **d'éviter une complexité en temps exponentielle** qui rend le programme inutilisable, sans pour autant saturer totalement la mémoire disponible.
- Cependant, ce n'est pas toujours le cas et il arrive que la complexité en mémoire augmente tellement que le compromis n'est plus envisageable, mais il est possible de réduire intelligemment l'espace mémoire occupé par de nombreux algorithmes dynamiques.

24

## VERSION RECURSIVE AVEC MÉMOÏSATION VS VERSION ITÉRATIVE BOTTOM UP?

- Faut-il favoriser la solution itérative bottom-up ou la solution recursive avec mémoïsation ?

Pas de réponse générale, et souvent le choix sera fait en fonction de critères variés, tous légitimes :

– L'architecture du reste du programme,
– le langage de programmation,
– la nature du problème,
– le gout du programmeur.

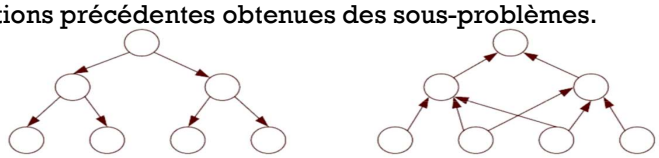
Quelques critères objectifs :

- La **solution bottom-up** est souvent légèrement plus rapide, car elle évite le (petit) surcout dû à la récursivité (gestion de la pile) et le test effectué à chaque appel pour vérifier si le résultat demandé a déjà été calculé.
- En revanche, pour certains problèmes, il peut arriver que l'on n'aura pas, finalement, besoin de tous les résultats correspondants à des sous-problèmes plus petits. Dans ce cas, **l'utilisation de la mémoïsation permet d'éviter de calculer des résultats inutiles**.

25

## PROGRAMMATION DYNAMIQUE-DIVISER ET RÉGNER

- La programmation dynamique est similaire à la méthode diviser et régner en ce sens que, une solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes.



diviser et régner — programmation dynamique

**La différence significative** entre ces deux méthodes est que:

- la programmation dynamique permet aux sous-problèmes de se superposer. Autrement dit, un sous-problèmes peut être utilisé dans la solution de deux sous-problèmes différents.
- A contrario, l'approche diviser et régner crée des sous-problèmes complètement séparés qui peuvent être résolus indépendamment l'un de l'autre.
- => La différence fondamentale entre ces 2 méthodes devient claire:

les sous-problèmes dans la programmation dynamique peuvent être en interaction,

alors dans la méthode diviser et régner, ils ne le sont pas.

26

## CARACTÉRISATION DE LA PROGRAMMATION DYNAMIQUE

Un **problème relève de la programmation dynamique** si :

- Le problème peut être résolu à partir de sous-problèmes similaires mais plus petits.
- Si l'on développe l'arbre des appels récursifs, un même sous-problème apparaît un grand nombre de fois.
- L'ensemble des sous-problèmes est discret, c'est-à-dire qu'on peut les indicer et ranger les résultats dans un tableau.

Ces 3 caractéristiques sont apparentes dans l'équation récursive qui caractérise le problème.
C'est, en général, **trouver la bonne équation récursive** qui constitue l'étape complexe et intelligente de la résolution du problème.

27

## EXERCICE APPLICATIF

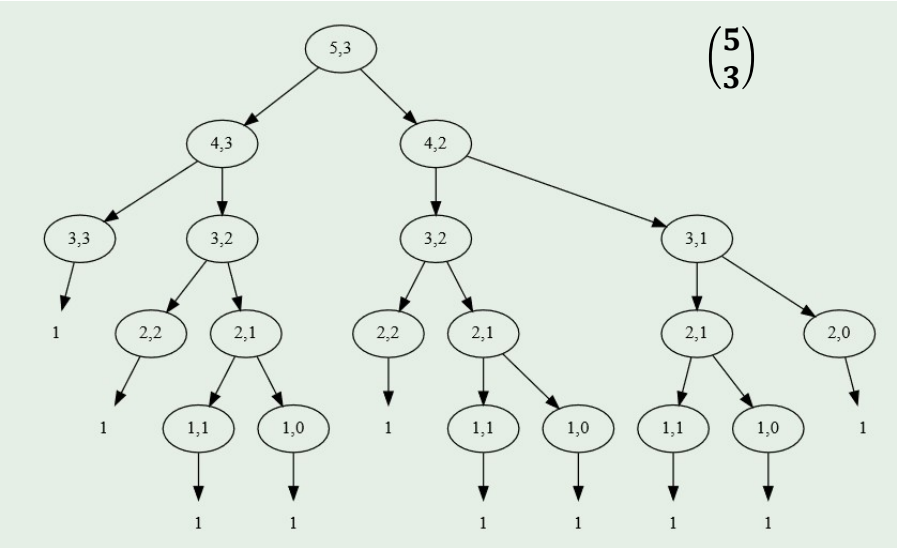
- Ecrire un algorithme qui permet de déterminer le nombre de combinaisons possibles de k éléments parmi n sachant que :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$
$$\binom{n}{0} = 1$$
$$\binom{n}{n} = 1$$

- Proposer d'abord une **approche naïve**,
- puis utiliser la programmation dynamique pour rendre l'algorithme plus efficace :
  - d'abord par **mémoïsation**
  - Puis par **approche itérative ascendante**

28

## ARBRE DES APPELS




29

## EXERCICE APPLICATIF

Approche naïve

```
# approche naive - Complexite : exponentielle O(2**n)
def Combinaison(n,k) :
    """Version récursive naïve"""
    if (k == 0) or (k == n) :
        return 1;
    else :
        return(Combinaison(n-1,k-1) + Combinaison(n-1,k))
#test
print( Combinaison(5,2))
```

30

## EXERCICE APPLICATIF

Approche en programmation dynamique (mémoisation)

```
# approche Programmation dynamique avec memoisation -
 Complexite :  O(nk)
# Avec une liste de listes
def CombinaisonMem(n,k, mat) :
  if (k == 0) or (k == n) :
    return 1;
  else :
    if mat[n][k]>0 :
      return mat[n][k]
    else :
      mat[n][k]=CombinaisonMem(n-1,k-1,mat) + CombinaisonMem(n-1,k,mat)
      return mat[n][k]
Complexite : O(n*k)
n=5
k=2
mat = [[ 0 for j in range (k+1)] for i in range (n+1)]
print( "mem", CombinaisonMem(n,k, mat))
# ou avec une fonction
def Combi(n,k):
    mat = np.zeros((n+1,k+1), int)
    return CombinaisonMem(n,k, mat)
print( "mem:", Combi(n,k))
```

## EXERCICE APPLICATIF

Approche en programmation dynamique (mémoisation avec un array)

```
# approche Programmation dynamique avec memoisation Complexite :  O(nk)
def CombinaisonMem(n,k, mat) :
    if (k == 0) or (k == n) :
      return 1;
    else :
      if mat[n,k]>0 :
        return mat[n,k]
      else :
        mat[n,k]=CombinaisonMem(n-1,k-1,mat) + CombinaisonMem(n-1,k,mat)
        return mat[n,k]
n=5
k=2
mat = np.zeros((n+1,k+1), int) # matrice resultante
print( "mem", CombinaisonMem(n,k, mat))
# ou avec une fonction
def Combi(n,k):
    mat = np.zeros((n+1,k+1), int)
    return CombinaisonMem(n,k, mat)
print( "mem:", Combi(n,k))
```

32

Approche en programmation dynamique itérative (Bottom-Up)

Exemple de remplissage de la matrice pour n=5 k=2

```
[[0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

0
```
[[1 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

1
```
[[1 0 0]
 [1 1 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

2
```
[[1 0 0]
 [1 1 0]
 [1 2 1]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

3
```
[[1 0 0]
 [1 1 0]
 [1 2 1]
 [1 3 3]
 [0 0 0]
 [0 0 0]]
```

4
```
[[1 0 0]
 [1 1 0]
 [1 2 1]
 [1 3 3]
 [1 4 6]
 [0 0 0]]
```

5
```
[[ 1  0  0]
 [ 1  1  0]
 [ 1  2  1]
 [ 1  3  3]
 [ 1  4  6]
 [ 1  5 10]]
```

# EXERCICE APPLICATIF

Approche en programmation dynamique itérative (Bottom-Up)

```
# approche Programmation dynamique itérative - Complexite :  O(nk)
def Combinaison_Dynamique(n,k):   #avec une liste de listes
    mat_res = [[ 0 for j in range (k+1)] for i in range (n+1)]
    for i in range (n+1):
        mat_res[i][0] = 1
        for j in range (1,k+1):
            mat_res[i][j] = mat_res[i -1][j] + mat_res[i -1][j -1]
    return mat_res[n][k]
print( "iter", Combinaison_Dynamique(5,2))
#Complexité de la solution: O(nk)
```

```
def Combinaison_Dynamique(n,k):# avec une matrice (tableau numpy)
    mat_res = np.zeros((n+1,k+1), int)
    for i in range (n+1):
        mat_res[i,0] = 1
        for j in range (1,k+1):
            mat_res[i,j] = mat_res[i -1,j] + mat_res[i -1,j -1]
    return mat_res[n,k]
print( "iter", Combinaison_Dynamique(5,2))
#Complexité de la solution: O(nk)
```

# EXEMPLES D'UTILISATION

Exemple classiques d'utilisation de la programmation dynamique:

- Problème du rendu de monnaie.
- Problème du sac à dos : nombre maximum d'objets qu'on peut mettre dans un récipient de taille fixée
- Répartition de plusieurs tâches entre plusieurs machines ou salles pour minimiser le temps total d'exécution.
- Placement des parenthèses lors du produit de plusieurs matrices pour minimiser le nombre de multiplications de coefficients effectué.
- Calcul de la « ressemblance » entre deux séquences d'ADN ou 2 mots.
  À chaque fois, le point est de définir les bons sous-problèmes, et d'identifier une relation de récurrence entre le problème initial ses sous-problèmes.