PGR. Programmation dynamique

I/ Introduction

La **programmation dynamique** est une méthode algorithmique utilisée pour résoudre des problèmes d'optimisation, en particulier ceux qui présentent une structure de sous-problèmes imbriqués ou une nature récursive. Elle date des années 1950, impulsée par Richard Bellman (oui, oui, celui de l'algorithme de Bellman-Ford utilisé par le protocole RIP).
Elle s'ajoute au principe **diviser pour régner** (dichotomie, tri fusion …) et aux **algorithmes gloutons**.

Pour comprendre son intérêt, prenons un exemple classique avec la *suite de Fibonacci* : le principe est de déterminer le nombre suivant en additionnant les deux précédents, sachant que les deux premiers valent respectivement 1 et 1.

Les cinq premiers termes de la suite de Fibonacci

Rang 0	Rang 1	Rang 2	Rang 3	Rang 4
0	1	1 + 0 = 1	1 + 1 = 2	2 + 1 = 3

On peut calculer les termes successifs de cette suite grâce à une fonction récursive que voici :

```
1  def fib(n) :
2      # Cas d'arrêt (pour n= 0 ou 1)
3      if n < 2 :
4          return n
5      # Cas général
6      else :
7          return fib(n-1) + fib(n-2)
8
9
10 print(fib(4))    # Attendu : 3
11
```

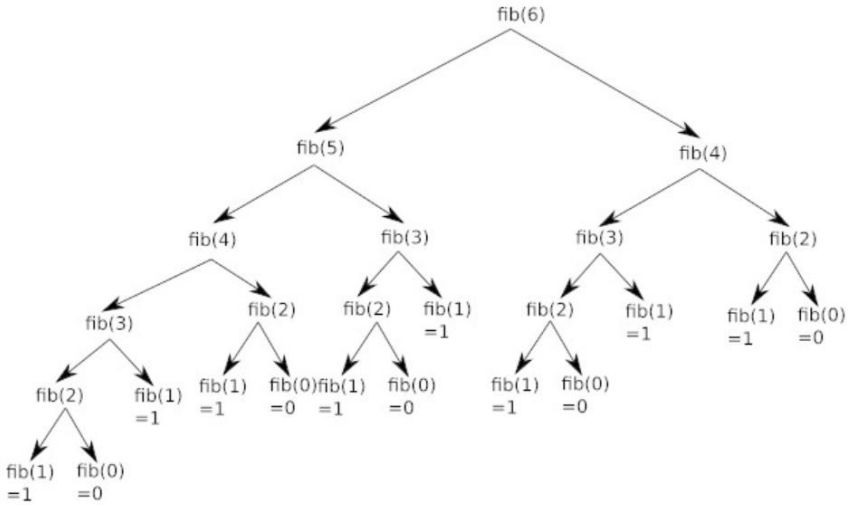
```
3
```

Mais …

```
In [*]:  1  print(fib(100)) # Surprise ...
```

Cela « mouline », aucun résultat ne s'affiche et un crash à la fin (dépassement de la capacité de la pile).

Pourquoi un ordinateur ne serait-il pas capable de calculer une suite simple au rang 100 ? Pour comprendre, voici, sauf forme d'arbre, la série d'appel de cette fonction pour n=6 :



Pour n = 6, on remarque bien que l'on trouve *fib(6) = 8* en additionnant les valeurs des feuilles.

On voit aussi qu'il y a beaucoup de calculs redondants :

- *fib(4)* est répété 2 fois,
- *fib(2)* l'est 5 fois !

On comprend intuitivement mieux pourquoi on ne parvient pas à calculer *fib(100)*, la complexité étant exponentielle.

Une idée serait de stocker les valeurs de fib(n) intermédiaires pour éviter de les recalculer. Voici un programme modifié en ce sens :

```python
def fib(n):
    # Cas d'arrêt
    if n < 2:
        return n
    # Cas général
    else :
        # Calcul du rang si nécessaire
        # et stockage dans le dictionnaire
        if n not in memo :
            memo[n] = fib(n-1) + fib(n-2)
        # Renvoi simple de la clé associée au rang
        return memo[n]


# Stocke les valeurs au fur et à mesure dans un dictionnaire
# les clés sont les rangs
memo = {}
```

Et là, pas de mauvaise surprise :

```python
print(fib(6))
print(fib(100))
```

```
8
354224848179261915075
```

Il s'agit d'un des aspects de la **programmation dynamique** qui permet notamment d'éviter les redondances de calculs en les stockant dans une structure (tableau, dictionnaire etc.).

II/ Programmation dynamique

1/ Introduction

La **programmation dynamique** suit deux approches principales :

- **Descendante ou Top-down** (ou "Mémoïsation") : Cette approche commence par résoudre le problème initial et s'appuie sur la résolution récursive des sous-problèmes. Elle utilise une structure de données (comme un tableau ou un dictionnaire) pour stocker les résultats des sous-problèmes déjà résolus, évitant ainsi les calculs redondants. C'est l'exemple précédent.
- **Ascendante ou Bottom-up** (ou "Tableau") : Cette approche consiste à résoudre d'abord les sous-problèmes les plus simples et à progresser vers la résolution du problème initial. Elle construit une table de solutions aux sous-problèmes en ordre croissant de complexité, en utilisant les résultats précédents pour résoudre les problèmes suivants.

La programmation dynamique est particulièrement efficace pour résoudre des problèmes comme la recherche du plus court chemin, la multiplication de chaînes de matrices, la plus longue sous-séquence commune, le sac à dos et bien d'autres. En exploitant la structure des problèmes et en évitant les calculs redondants, la programmation dynamique permet de réduire considérablement le temps de calcul et la complexité des algorithmes.

On la retrouve aussi dans le domaine de **l'économie** et la **logistique** (gestion de projets, trafic routier, gestion du flux de clients, tâches de planification, gestion des investissements en fonction des attendus, risques etc.).

2/ L'approche descendante

Elle est appelée aussi « mémoïsation ». L'idée est de stocker la solution de sous-problèmes intermédiaires apparaissant lors de la résolution du problème initial.

Voici un exemple avec le problème classique du *rendu de monnaie*. L'année dernière, on a vu que l'on pouvait se servir d'un **algorithme glouton** pour trouver une solution, optimale si le système monétaire est canonique (pas comme celui des Anglais jusque dans les années 1970 !).
Lien vers le cours de Première :
https://github.com/lmayer65/NSI_T/blob/main/Programme_NSI_1/Algorithmes/AGR.Algorithme.Glouton.pdf

Le principe est tout simplement de **rendre** à chaque fois **la pièce ayant la plus forte valeur possible**.

Ainsi, pour rendre (dans notre système monétaire), 66 cts, on agit ainsi :

Reste à rendre	66	16	6	1
Pièce rendue	50	10	5	1
Nbre de pièces rendues	1	2	3	4

On voit qu'avec 4 pièces, on a rendu la monnaie et ce de manière optimale.

Voici une fonction récursive de l'algorithme glouton :

```
def glouton_rec(pieces, reste_a_rendre, rendu = []) :
    # Cas d'arrêt
    if reste_a_rendre == 0 :
        return rendu

    # Impossible ici de rendre la monnaie, le reste  rendre est inférieur
    # aux pièces disponibles.
    if reste_a_rendre < pieces[len(pieces) - 1] :
        return f"""Rendu impossible, il reste {reste_a_rendre} à rendre,
pas de pièces disponibles de ce montant"""

    # Cas général
    else :
        for piece in pieces :
            # Si on trouve la pièce de la valeur la plus grande possible
            if reste_a_rendre >= piece :
                rendu.append(piece)  # Ajout au rendu
                return glouton_rec(pieces, reste_a_rendre - piece, rendu)
```

Jeu de test ici :

```
# Jeu de test
pieces = [50,20,10,5,2,1]
reste_a_rendre = 66
rendu = []
print(glouton_rec(pieces, reste_a_rendre, rendu))
```

```
[50, 10, 5, 1]
```

On retrouve bien le résultat attendu

On considère maintenant le système monétaire suivant *pièces = [50,10,5,3,2]* et on souhaite rendre 11 cts.
On voit bien qu'il suffit de rendre 1 pièce de 5 centimes et 2 pièces de 3 centimes soit *rendu = [5,3,3].*

Que trouve l'algorithme glouton ? Eh bien …

```
# Jeu de test
pieces = [50,20,10,5,3,2]
reste_a_rendre = 11
rendu = []
print(glouton_rec(pieces, reste_a_rendre, rendu))
```

```
Rendu impossible, il reste 1 à rendre,
pas de pièces disponibles de ce montant
```

L'algorithme glouton est mis en défaut et pourtant, la solution existe. Il se peut aussi qu'il ne trouve pas la solution optimale (système monétaire non canonique) par exemple.
La raison est dans le fonctionnement même de ce type d'algorithme : une fois un sous-problème traité (de manière optimale), il n'y revient jamais.

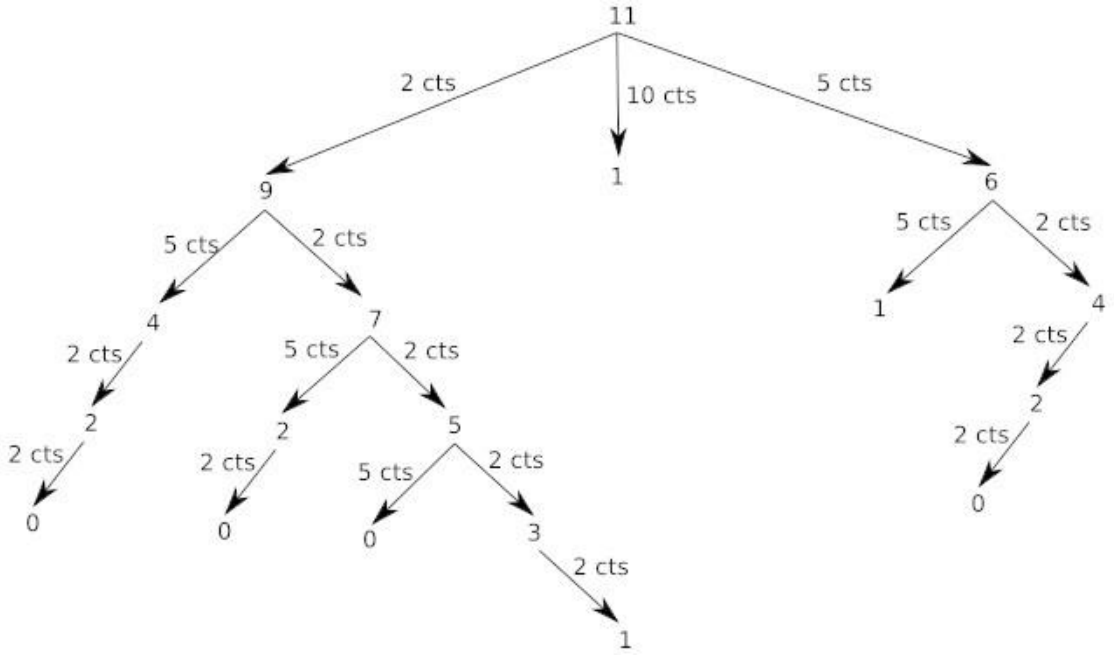
La programmation dynamique va permettre de traiter ces sous-problèmes, en prenant en compte les conséquences sur les traitements futurs : on peut donc « revenir en arrière ».

Pour traiter ce problème de rendu de monnaie, on peut chercher une relation de récurrence, c'est-à-dire un cas élémentaire de base et le cas général, permettant de simplifier le problème.

Voici une possibilité :
Cas élémentaire : la somme à rendre est nulle, aucune pièce à rendre donc.
Cas général : le nombre de pièces total à rendre est égal à la somme du minimum de pièces données pour la somme ôtée de cette pièce et de un, correspondant à la pièce rendue.

Exemple : on souhaite rendre la somme de 11 cts dans ce système de monnaie : pièces = [10,5,2]
Le cas élémentaire est évident, lorsque la monnaie est rendue, il n'y a plus de pièces à rendre.

Pour le cas général, voici le principe :



Voici un exemple de chemin (celui à gauche) :

- Le nombre de pièces à rendre sur 11 cts est celui à rendre sur 9 cts plus une pièce (celle de 2 cts).
- Le nombre de pièces à rendre sur 9 cts est celui à rendre sur 4 cts plus une pièce (celle de 5 cts).
- Le nombre de pièces à rendre sur 4 cts est celui à rendre sur 2 cts plus une pièce (celle de 2 cts).
- Le nombre de pièces à rendre sur 2 cts est celui à rendre sur 2 cts plus une pièce (celle de 2 cts).
- Cas élémentaire, il n'y a plus de pièces à rendre. Il faut donc en tout 4 pièces.

En réalité, il s'agit ici de trouver la feuille ayant **la hauteur minimale**.

On remarquera que tous les cas sont traités, on peut parler de **force brute**. De plus, on constate que l'on peut revenir en arrière : ainsi, avoir reçu une pièce de 2 cts n'empêche pas d'envisager d'en recevoir plutôt une de 10 cts ou de 5 cts (au niveau 1 de l'arbre).

Voici un exemple de programme :

```
def rendu_monnaie_rec(pieces,reste_a_rendre):
    # Cas de base
    if reste_a_rendre == 0:
        return 0
    # Cas général
    else :
        # Nombre de pièces par défaut : infini
        # on est sûr de trouver mieux
        nb_mini = float('inf')

        for i in range(len(pieces)) :
            # Si une pièce convient, on la rend
            if pieces[i] <= reste_a_rendre :

                # Relation de récurrence
                nb_pieces = 1 + rendu_monnaie_rec(pieces, reste_a_rendre - pieces[i])

                # Si on trouve un meilleur rendu (moins de pièces)
                if nb_pieces < nb_mini:
                    nb_mini = nb_pieces

        return nb_mini
```

Jeu de test ici :

```
# Jeu de test
pieces = [50,20,10,5,3,2]
reste_a_rendre = 11
print(f"""Nombre de pièces minimal à rendre :
    {rendu_monnaie_rec(pieces, reste_a_rendre)}""")
```

```
Nombre de pièces minimal à rendre :
    3
```

On trouve bien 3 pièces.

Malheureusement, on retrouve le même problème qu'avec la suite de Fibonacci : le dépassement de la capacité de la pile se reproduit très rapidement.

L'étoile indique que le programme mouline …

```
In [*]: # Jeu de test
        pieces = [50,20,10,5,3,2]
        reste_a_rendre = 121
        print(f"""Nombre de pièces minimal à rendre :
            {rendu_monnaie_rec(pieces, reste_a_rendre)}""")
```

Pour corriger ce problème et éviter les redondances, on va appliquer le principe de **programmation dynamique** (descendante ici) en stockant les calculs déjà effectués.

Voici un exemple de programme :

```
def rendu_monnaie_desc(pieces,reste_a_rendre):
    # Cas de base
    if reste_a_rendre == 0:
        return 0
    # Cas général
    else :
        # Si on a déjà calculé la valeur, on la renvoie simplement
        if reste_a_rendre in memo :
            return memo[reste_a_rendre]
        else:
            nb_mini = float('inf')

            for i in range(len(pieces)):
                if pieces[i] <= reste_a_rendre:
                    nb_pieces = 1 + rendu_monnaie_asc(pieces,reste_a_rendre - pieces[i])
                    if nb_pieces < nb_mini:
                        nb_mini = nb_pieces
                        # Stockage du résultat
                        memo[reste_a_rendre] = nb_mini

            return nb_mini
```

Jeu de test concluant

```
# Jeu de test
pieces = [50,20,10,5,3,2]
reste_a_rendre = 121
print(f"""Nombre de pièces minimal à rendre :
    {rendu_monnaie_asc(pieces, reste_a_rendre)}""")
```

```
Nombre de pièces minimal à rendre :
    6
```

3/ L'approche ascendante

Dans la méthode **ascendante**, on va "de bas en haut" : il s'agit du sens contraire à celui utilisé par les méthodes récursives. On commence par calculer des solutions pour les sous-problèmes les plus petits, puis, de proche en proche, on calcule les solutions des problèmes en utilisant le principe d'optimalité et on mémorise les résultats dans un tableau, le tout à l'aide boucles.

Voici un exemple de programme reprenant la suite de Fibonacci précédente :

On calcule tout simplement les termes en fonctions des deux précédents qui sont stockés dans une liste.

```python
def fib(n):
    # Initialisation du tabeau et valeurs de départ
    f = [0]*(n+1)
    f[1] = 1

    # Détermination de chaque terme en fonction des deux précédents
    for i in range(2, n+1):
        f[i] = f[i-1] + f[i-2]

    return f[n]

print(fib(10))
```

```
55
```

Représentation de l'enchaînement des calculs pour fib(5) avec mémorisation des résultats intermédiaires.

Calcul de fib(5) : Enchaînement des calculs avec mémorisation des résultats à chaque étape					
0	1	2	3	4	5
mémo. ↓	mémo. ↓	calcul + mémo. ↓	calcul + mémo. ↓	calcul + mémo. ↓	calcul + mémo. ↓
f[0]	f[1]	f[2] = f[1] + f[0]	f[3] = f[2] + f[1]	f[4] = f[3] + f[2]	f[5] = f[4] + f[3]

On constate que la complexité est ici linéaire.

Et pour le rendu de monnaie ? L'approche est la même, on part **des cas de bases** pour « remonter » au **cas général**.

Si on reprend le système de monnaie *pièces = [1,2,5,10,20,50]*, voici le principe :

- pour rendre 0 ct, on ne rend aucune pièce,
- pour rendre 1 ct, on rend 1 pièce,
- pour rendre 2 cts, on rend 1 pièce (de 2 cts),
- pour rendre 3 cts, on rend 2 pièces (un de 2 cts et une de 1 ct) etc.

Toutes ces valeurs seront stockées dans un tableau de taille *n+1* (à cause du 0) avec *n* correspondant à la somme à rendre.

Voici un programme qui suit cette logique :

```python
def rendu_monnaie_asc(pieces,somme_a_rendre):
    # Tableau stockant le nombre de pièces à rendre
    tab = [float('inf')] * (somme_a_rendre+1)
    tab[0] = 0 # Valeur initiale

    # Parcours de tous les indices du tableau de 0 à
    # somme_a_rendre
    for a_rendre in range(1, somme_a_rendre+1):
        # Pour chaque indice, parcours de toutes les pièces disponibles
        for piece in pieces:
            if a_rendre >= piece:
                tab[a_rendre] = min(tab[a_rendre], 1 + tab[a_rendre-piece])

    # Si on trouve une solution
    if tab[somme_a_rendre] != float('inf') :
        return tab[somme_a_rendre]
```

Jeu de tests

```python
# Jeu de test
pieces = [50,20,10,5,2,1]
reste_a_rendre = 121
print(f"""Nombre de pièces minimal à rendre :
    {rendu_monnaie_asc(pieces, reste_a_rendre)}""")
```

```
Nombre de pièces minimal à rendre :
    4
```

Si on fait tourner le programme, voilà ce qui se passe :

- tab[0] vaut 0,
- tab[1] = min(tab[1], 1 + tab[1 – 1]) = min(infini, 1 + 0) = 1 (pour un rendu de 1 ct)
  On ne peut plus rien tester, les autres pièces ont un montant trop élevé.
- tab[2] = min(tab[2], 1 + tab[2 – 1]) = min(infini, 1 + tab[1]) = 2 (pour un rendu de 1 ct)
- tab[2] = min(tab[2], 1 + tab[2 – 2]) = min(2, 1 + tab[0]) = 1 (pour un rendu de 2 ct)
  On ne peut plus rien tester, les autres pièces ont un montant trop élevé.

- tab[3] = min(tab[3], 1 + tab[3 – 1]) = min(infini, 1 + tab[2]) = 2 (pour un rendu de 1 ct)
- tab[3] = min(tab[3], 1 + tab[3 – 2]) = min(2, 1 + tab[1]) = 2 (pour un rendu de 2 ct)
  On ne peut plus rien tester, les autres pièces ont un montant trop élevé.
- tab[4] = min(tab[4], 1 + tab[4 – 1]) = min(infini, 1 + tab[3]) = 3 (pour un rendu de 1 ct)
- tab[4] = min(tab[4], 1 + tab[4 – 2]) = min(3, 1 + tab[2]) = 2 (pour un rendu de 2 ct)
  On ne peut plus rien tester, les autres pièces ont un montant trop élevé.
- tab[5] = min(tab[5], 1 + tab[5 – 1]) = min(infini, 1 + tab[4]) = 3 (pour un rendu de 1 ct)
- tab[5] = min(tab[5], 1 + tab[5 – 2]) = min(3, 1 + tab[3]) = 3 (pour un rendu de 2 ct)
- tab[5] = min(tab[5], 1 + tab[5 – 5]) = min(3, 1 + tab[0]) = 1 (pour un rendu de 5 ct)
- *On ne peut plus rien tester, les autres pièces ont un montant trop élevé.*
  etc.

La complexité est ici de l'ordre du produit de nombre de pièces disponibles par la somme à rendre.

4/ Programmation dynamique, algorithmes gloutons et « diviser pour régner »

La programmation dynamique, l'approche diviser pour régner et les algorithmes gloutons sont des techniques d'optimisation algorithmique qui peuvent être utilisées pour résoudre des problèmes. Voici les différences principales entre ces techniques :

- **La programmation dynamique** est généralement utilisée pour les problèmes d'optimisation combinatoire, où une solution doit être trouvée parmi un grand nombre de possibilités. La technique consiste à décomposer le problème en sous-problèmes plus petits et à résoudre chaque sous-problème -dépendant souvent des autres- en stockant les résultats : un sous-problème traité ne le sera donc pas une seconde fois.
- La **technique « diviser pour régner »** est souvent utilisée pour les problèmes où une solution peut être facilement trouvée en combinant les solutions de sous-problèmes plus petits et indépendants. La technique consiste à diviser le problème en sous-problèmes plus petits, à résoudre chaque sous-problème de manière récursive et à combiner les résultats pour obtenir la solution globale. Contrairement à la programmation dynamique, la technique de la diviser pour régner ne nécessite pas de mémorisation car chaque sous-problème est résolu indépendamment.
- Les **algorithmes gloutons** sont une technique d'optimisation qui consiste à faire des choix locaux optimaux à chaque étape, dans l'espoir que ces choix locaux conduiront à une solution globale optimale. Les algorithmes gloutons sont souvent plus simples et plus rapides que les techniques de programmation dynamique et de diviser pour régner. Cependant, les algorithmes gloutons ne garantissent pas toujours une solution optimale globale. Les algorithmes gloutons sont souvent utilisés dans des problèmes tels que la recherche du chemin le plus court dans un graphe pondéré, le problème du sac à dos, ou encore le problème du rendu de monnaie.

Bien qu'elles soient différentes dans leur façon de traiter les problèmes, elles partagent certaines relations et parfois des points communs :

- Objectif commun : Les trois approches visent à résoudre des problèmes complexes en décomposant ou en simplifiant le problème initial, afin de réduire la complexité temporelle et/ou spatiale.
- Adaptabilité : Les trois techniques peuvent être appliquées à divers problèmes d'optimisation et de recherche. Cependant, leur efficacité et leur applicabilité dépendent de la nature du problème et de ses contraintes.
- Approches complémentaires : Dans certains cas, les trois techniques peuvent être combinées pour résoudre un problème spécifique. Par exemple, un algorithme de programmation dynamique peut utiliser une approche de diviser pour régner pour réduire la taille des sous-problèmes, ou un algorithme glouton peut être utilisé pour guider une recherche plus approfondie.