Chapitre 10 : programmation dynamique

10.1 Principe et introduction

La programmation dynamique est une des techniques de programmation qui visent à réduire le temps de calcul d'une solution optimale. Elle peut aussi bien s'appliquer si le problème a une complexité polynomiale $O(n^k)$ que si le problème a une complexité exponentielle $O(k^n)$.
Elle est souvent appliquée à des problèmes d'optimisation où :

- il existe un très grand nombre de solutions. Il est donc coûteux, voire impossible, de calculer toutes les solutions pour choisir la meilleure.
- on recherche une solution optimale (il peut en exister plusieurs), par exemple celle dont le coût est minimal ou maximal

L'idée de base de la programmation dynamique est qu'une solution est composée de sous-solutions. Or, si nous sommes sûrs que la solution optimale ne dépend que de sous-solutions optimales, alors le problème peut être représenté de manière récursive et la recherche de la solution optimale ne nécessite que le calcul d'un nombre limité de sous-solutions.

Ce principe a été introduit par **Richard Bellmann** dans les années 50 et utilisé dans l'algorithme de Bellmann-Ford pour la recherche d'un plus court chemin.

La programmation dynamique tout comme la méthode « diviser pour régner » utilise des sous-solutions. Mais à la différence de cette méthode, un certain nombre de sous-solutions peuvent être rencontrées plusieurs fois dans la résolution globale. À des fins d'optimisation elles sont alors stockées en mémoire pour ne pas être recalculées inutilement plusieurs fois.

10.2 Méthode et algorithme

a. Exemple suite de Fibonacci

Considérons la suite des nombres de Fibonacci. Le principe est simple : on commence par les deux premiers nombres 1 et 1 puis le suivant est la somme des deux précédents, soit 2 et on continue de même

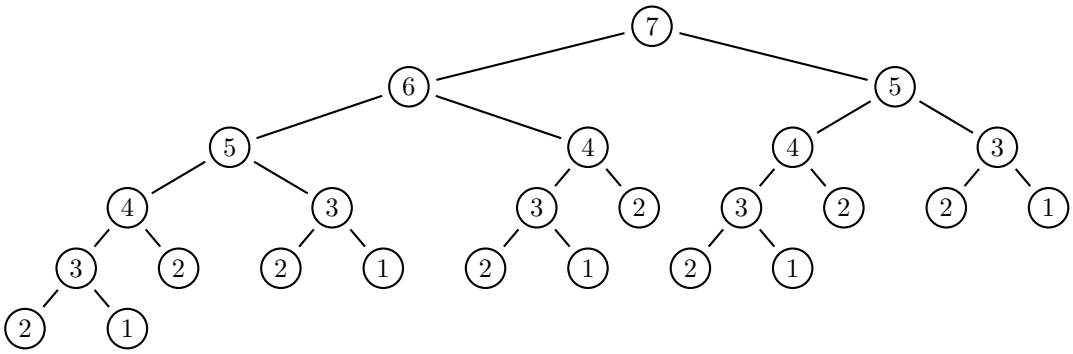
n	1	2	3	4	5	6	7	
fibo	1	1	2	3	5	8	13	...

▶ Exercice 1

Saisir cet alogrithme sur le site https ://pythontutor.com pour évaluer le nombre d'appels récursifs effectués pour calculer `fib_rec(7)`

```
def fibo_rec(n):
        if n <= 2 :
            return 1
        else:
            return fibo_rec(n-1) + fibo_rec(n-2)
```

En réprésentant les appels récursifs sous forme d'arbre binaire, on se rend compte que certains appels sont redondants.



Par exemple, le calcul de `fib_rec(5)` doit être fait à deux reprises, celui de `fib_rec(4)` à trois reprises et ainsi de suite. Il serait avantageux en terme de nombre d'opérations de stocker en mémoire chaque résultat pour le réutiliser si nécessaire.

▶ Exercice 2 Fibonacci avec cache

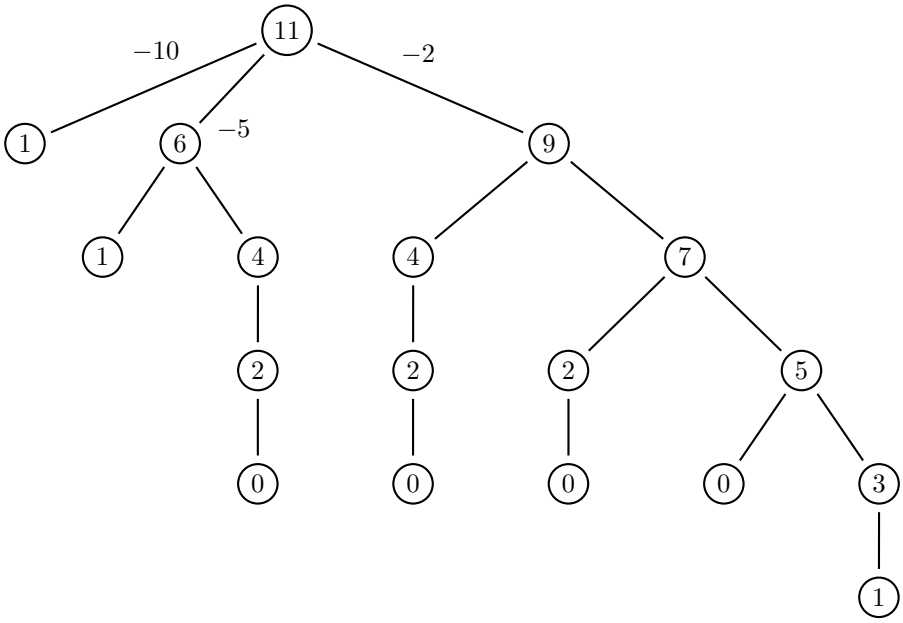
On peut utiliser un dictionnaire pouvant simuler une sorte de mémoire cache. Saisir cet alogrithme sur le site https ://-pythontutor.com et visualiser la différence de fonctionnement avec le script précédent.

```
fibocache = {1 : 1, 2 : 1 }
def fib_rec_dyn(n):
    if n in fibocache.keys():
        return fibocache[n]

    value =  fib_rec_dyn(n-1) + fib_rec_dyn(n-2)
    fibocache[n] = value

    return value
```

b. Exemple rendu de monnaie

Ce problème est un classique et il a déjà été vu en classe de première, avec l'utilisation de la méthode dite gloutonne. Pour rappel, cette méthode rend la plus grande pièce possible à chaque étape en espérant qu'un choix optimal local à chaque étape aboutisse au final à une solution optimale globale.
L'optimalité dans ce cas consiste à rendre le moins de pièces possible. On a vu que l'optimalité de la méthode gloutonne n'est pas assurée pour le probléme de rendu de monnaie, en effet en fonction du système de monnaie la solution gloutonne peut ne pas être optimale.

Considérons que l'on dispose uniquement de pièces de 10 centimes, de 5 centimes et de 2 centimes (on ne dispose malheureusement pas de pièces de 1 centime) et que l'on doive rendre 11 centimes.
La méthode gloutonne n'aboutira pas dans ce cas, puisqu'elle commencera par rendre une pièce de 10, il restera 1 centime dont on ne dispose pas.
Essayons d'envisager toutes les possibilités, on utilisera encore une fois un arbre.



Il existe plusieurs solutions optimales pour lesquelles on rend quatre pièces et il existe des chemins qui n'aboutissent pas. On se rend compte aussi que certains sous-arbres se répètent plusieurs fois (la séquence 4 -> 2 -> 0). On peut appliquer les principes de la programmation dynamique. En effet dans le cas de plus grande somme de monnaie à rendre, l'arbre sera beaucoup plus grand et les redondances plus fréquentes que sur cet exemple.

▶ Exercice 3 Rendu de monnaie récursif

Le programme ci-dessous permet de calculer de manière récursive le nombre minimum de pièces qu'il faudra rendre.

```
def rendu_rec(P,somme):
    if somme == 0 :
     return 0
    else :
        minimum = 1000
    #on teste toutes les possibilites
        for piece in P:
            if piece <= somme:
                nb_pieces = 1 + rendu_rec(P, somme - piece)
                if nb_pieces < minimum:
                        minimum = nb_pieces
        return minimum
```

Tester le programme avec des pièces de 10, 5 et 2 centimes pour rendre 11 centimes

```
>>>rendu_rec([10,5,2],11)
```

Tester avec les mêmes pièces pour rendre 72 centimes. Que constate-t-on ?
On va tenter d'appliquer la principe de la programmation dynamique à ce problème, en modifiant légèrement le script précédent :

```
rendu_dico = {0:0}

def rendu_rec_dyn(P,somme):
    if somme in rendu_dico.keys() :
     return ............
    else :
        minimum = 1000
    #on teste toutes les possibilites
        for piece in P:
            if piece <= somme:
                nb_pieces = 1 + rendu_rec_dyn(P, somme - piece)
                if nb_pieces < minimum:
                        minimum = nb_pieces
        ..........................
        return minimum
```

Comparer les vitesses des exécutions des deux différents scripts sur les mêmes exemples, que constate-t-on ?

NB : le langage Python possède des techniques avancées qui ne sont pas au programme de terminale, permettant de mettre automatiquement en cache des appels récursifs (voir du côté du décorateur de fonction Python `@lru_cache` ou `@cache` en python 3.10)

```
from functools import lru_cache

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