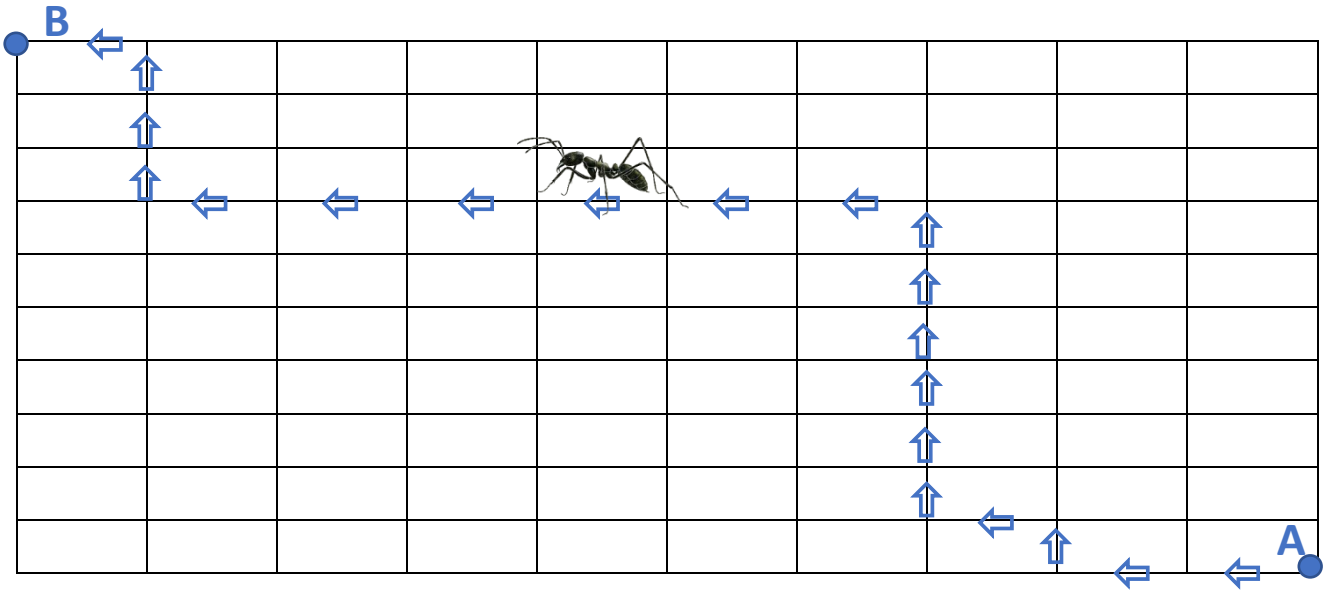
Chapitre 21 - Programmation dynamique

En informatique, la programmation dynamique est une méthode algorithmique qui permet de résoudre des problèmes d'optimisation efficacement.
Pour optimiser des problèmes de grande taille, pour lesquels une solution naïve serait trop gourmande en ressources, il est nécessaire d'écrire son code en changeant de paradigme. Parmi les paradigmes algorithmiques déjà vus, on peut citer l'algorithme glouton, les k plus proches voisins, diviser pour régner. On en voit ici un nouveau : la programmation dynamique. D'une manière générale, la méthode consiste « ***à diviser un problème complexe en sous-problèmes dont la taille varie durant l'exécution. Les résultats de ces sous-problèmes sont mémorisés afin de pouvoir être réutilisés ensuite.*** »

On voit dans ce chapitre plusieurs exemples concrets. On fait une synthèse de ces différentes situations en fin de chapitre.

1- PROBLEME DE LA FOURMI :

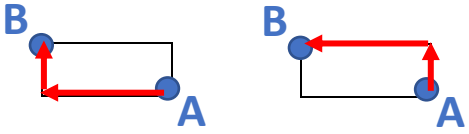
Alice pose le problème suivant à Basile. Elle dessine sur une feuille de papier une petite grille de N lignes et N colonnes. Par exemple, si $N = 10$, on obtient la figure suivante :



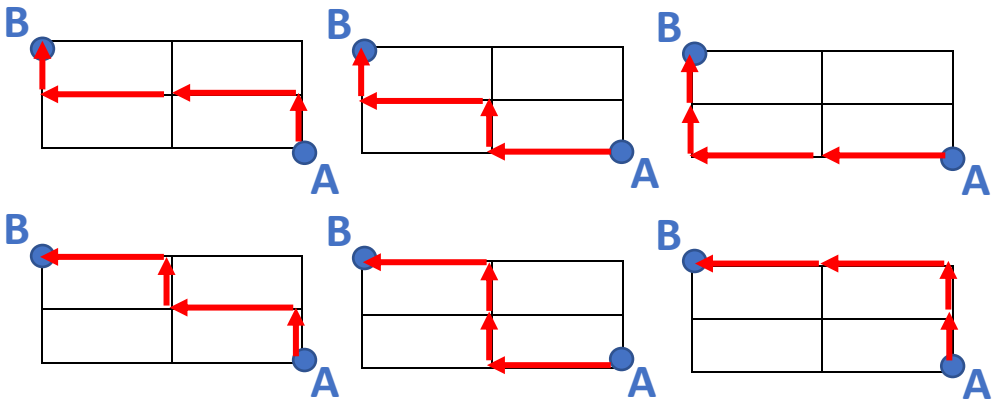
Une fourmi part du point **A** et veut aller sur le point **B**. Elle ne peut que suivre une ligne et se déplacer vers la gauche ou vers le haut, sans pouvoir revenir en arrière.

Question : **Combien de chemins différents, cette fourmi peut-elle emprunter ?**

1- Nombre de chemins pour $N = 1$:

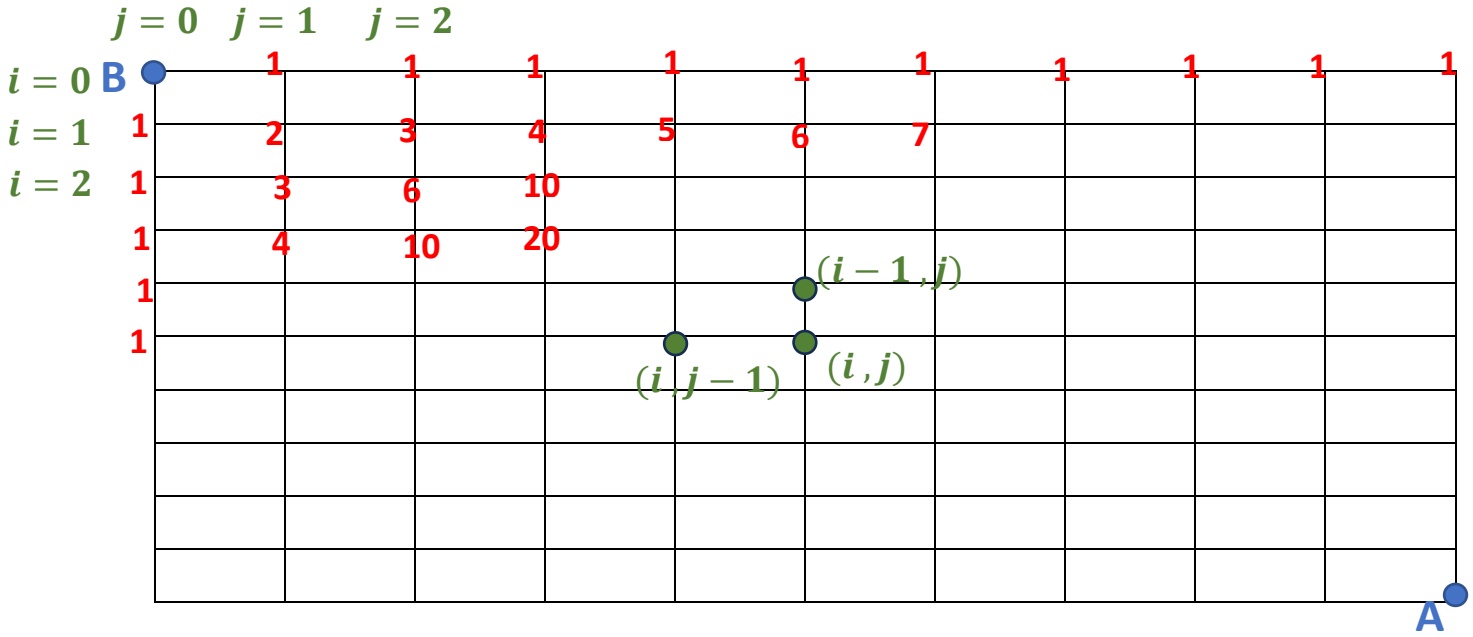


2- Nombre de chemins pour $N = 2$:



3- Nombre de chemins pour N quelconque :

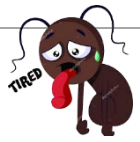
On se propose ici d'écrire un algorithme qui utilise le principe de la programmation dynamique. Cela consiste à résoudre le problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.



⇨ Ecrire le code de la fonction *fourmi()* ayant pour paramètre un entier naturel N et retournant le nombre de chemins possibles. On donne 2 exemples d'exécutions :

```
>>> fourmi(1)
2
```

```
>>> fourmi(2)
6
```



Version itérative :

```
def fourmi(N) :
    l = [[0 for j in range(N+1)] for i in range(N+1)]
    for i in range(N+1) :
        for j in range(N+1) :
            if i == 0 or j == 0:
                l[i][j] = 1
            else :
                l[i][j] = l[i][j-1] + l[i-1][j]
    return l[N][N]
```

Version récursive :

```
def fourmi_rec(i,j) :
    if i == 0 or j == 0 : return 1
    else :
        return fourmi_rec(i-1,j) + fourmi_rec(i,j-1)

def fourmi(N) :
    return fourmi_rec(N,N)
```

Analyse : On a ici un code qui reprend le principe de la programmation dynamique :

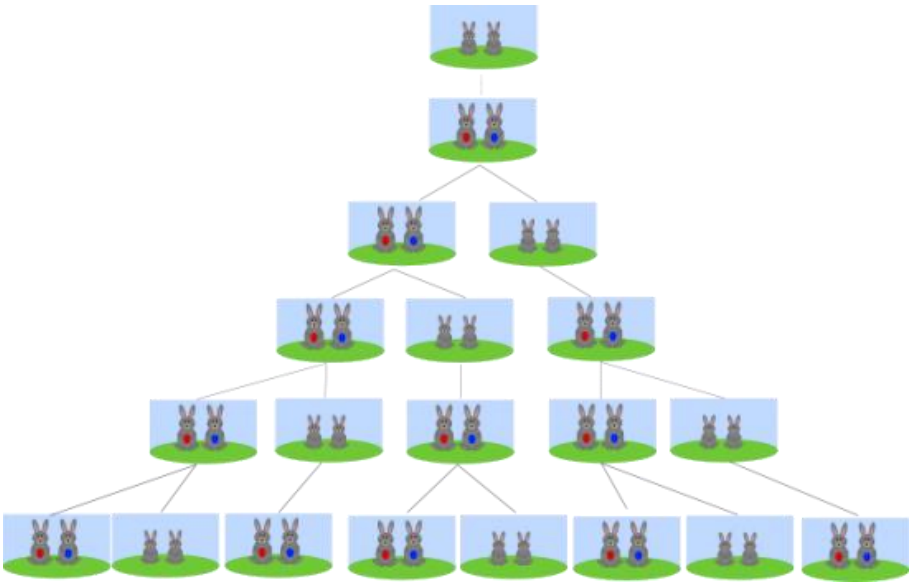
- On divise le problème complexe en sous-problèmes en partant du point d'arrivé.
- Les résultats de ces sous-problèmes sont mémorisés dans une liste afin de pouvoir être réutilisés ensuite.

2- PROBLEME DE LA SUITE DE FIBONACCI :

La suite de Fibonacci commence ainsi : $0\,,1\,,1\,,2\,,3\,,5\,,8\,,13\,,21\,,34\,,55\,,89\,,144\,,233\,,\ldots$
Ses deux premiers termes sont 0 et 1, et ensuite, chaque terme successif est la somme des deux termes précédents. Mathématiquement, elle est définie par récurrence :

$$F_0 = 0 \quad \text{et} \quad F_1 = 1.$$
Pour tout $n \geq 2$: $\boldsymbol{F_n = F_{n-2} + F_{n-1}}$

Son inventeur est Léonard de Pise (1175 − 1250), aussi connu sous le nom de Leonardo Fibonacci, qui a rapporté d'Orient la notation numérique indo-arabe et a écrit et traduit des livres influents de mathématiques. La suite de Fibonacci peut être considérée comme le tout premier **modèle mathématique** en dynamique des populations ! En effet, elle y décrit la croissance d'une population de lapins sous des hypothèses très simplifiées, à savoir : chaque couple de lapins, dès son troisième mois d'existence, engendre chaque mois un nouveau couple de lapins, et ce indéfiniment. Partons donc d'un couple de lapins le premier mois. Le deuxième mois, on n'a toujours que ce même couple, mais le troisième mois on a déjà 2 couples, puis 3 couples le quatrième mois, 5 couples le cinquième mois, etc. La croissance de cette population est belle et bien décrite par la suite de Fibonacci.



1- Algorithme récursif naïf qui calcule un terme F_n :

⇨ Ecrire le code python de la fonction *fibNaif()* qui a comme paramètre un entier naturel n et qui renvoie le terme F_n de la suite de Fibonacci. On donne 2 exemples d'exécutions :

```
>>> fibNaif(6)
8
```

```
>>> fibNaif(7)
13
```

⇨ Calculer les termes suivants et estimer « à la louche le temps de calcul » :

$F_{10} = 55$	$F_{20} = 6765$	$F_{30} = 832040$	$F_{40} = 102334155$	$F_{50} = ?$
Temps : instantané	Temps : instantané	Temps : $\approx 1s$	Temps : $\approx 45\ s$	Temps : très long, $> 5\ mn$

```python
def fibNaif(n):
    if n < 2  :
        return n
    else:
        return fibNaif(n-1) + fibNaif(n-2)
```

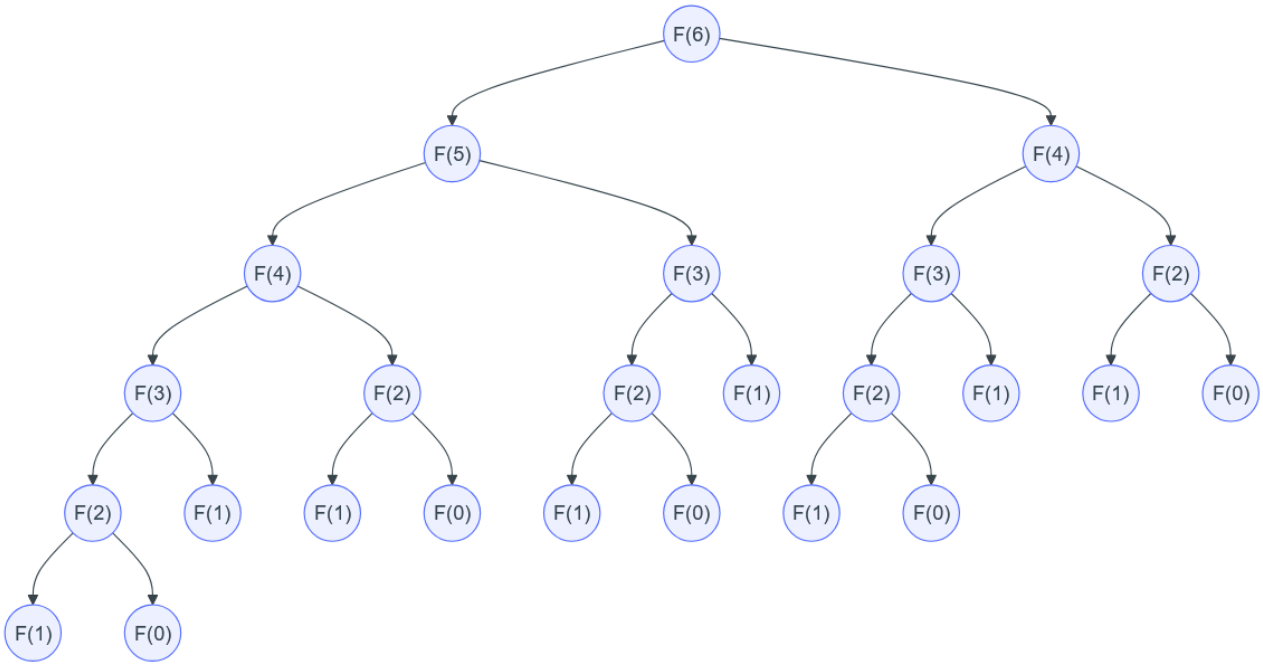
2- Algorithme récursif qui calcule un terme F_n en adoptant une méthode type programmation dynamique :

a) Pourquoi l'algorithme récursif naïf prend-il autant de temps ?

Pour calculer le terme de rang 6, il faut calculer celui de rang 5 et celui de rang 4
Pour calculer le terme de rang 5, il faut calculer celui de rang 4 et celui de rang 3
Pour calculer le terme de rang 4, il faut calculer celui de rang 3 et celui de rang 2
Pour calculer le terme de rang 3, il faut calculer celui de rang 2 et celui de rang 1
Pour calculer le terme de rang 2, il faut calculer celui de rang 1 et celui de rang 0.

On remarque que F_4 est calculé deux fois (une fois F_6 et une fois pour F_5) , F_3 est calculé pour chaque calcul de F_4 et F_5 (donc trois fois en tout) et que F_2 est calculé 5 fois en tout. On obtient même 13 appels à F_0 ou F_1. (On remarque d'ailleurs que $F_6 = 13$)

On peut représenter cela avec l'arbre des appels :



Plus on augmente le numéro de rang, plus il y a des calculs qui sont répétés et refaits.

b) Modification du code en reprenant les principes de la programmation dynamique :

Pour éviter d'exécuter le calcul d'un terme de rang k plusieurs fois, on peut le mémoriser dans une liste lorsque le premier calcul est exécuté. Les prochains calculs qui utiliseront cette valeur de F_k iront la récupérer dans cette liste. En procédant ainsi, on adopte une méthode de type programmation dynamique : « *on décompose le calcul global en sous-problèmes que l'on résout des plus petits aux plus grands, en stockant les résultats intermédiaires* ».

⇨ Ecrire le code python de la fonction *fibDyn()* qui a comme paramètre un entier naturel n et qui renvoie le terme F_n de la suite de Fibonacci. On donne 2 exemples d'exécutions :

```
>>> fibDyn(6)
8
```

```
>>> fibDyn(7)
13
```

```python
def fibDyn(n):
    memo = [None]*(n+1)
    return  fib(n, memo)


def fib(n,memo) :
    if memo[n] != None:
        return memo[n]
    elif n == 0 or n == 1:
        memo[n] = n
    else:
        memo[n] = fib(n-1,memo) + fib(n-2,memo)
    return memo[n]
```

⇨ Calculer les termes suivants et estimer « à la louche le temps de calcul » :

$F_{10} = 55$	$F_{50} =$ 12586269025	$F_{100} \approx 3.5\ 10^{19}$	$F_{500} \approx 1.4\ 10^{105}$	$F_{900} \approx 5.5\ 10^{188}$
Temps : instantané	Temps : instantané	Temps : instantané	Temps : instantané	Temps : instantané

Analyse : On a ici un code qui reprend le principe de la programmation dynamique : « *on décompose le calcul global en sous-problèmes que l'on résout des plus petits aux plus grands, en stockant les résultats intermédiaires* ». Cela le rend bien plus efficace que la version naïve.

Point Cours : **Mémoïsation**

Puisque qu'un algorithme naïf conduit à recalculer plusieurs fois une même valeur, on peut conserver en mémoire, au fur et à mesure, toutes les valeurs déjà calculées afin de ne pas recommencer le calcul. Ceci demande plus de mémoire mais permet un gain de temps.

La structure de donnée contenant les résultats mis en mémoire s'appelle le **cache**. **Mémoïser** une fonction consiste à la doter de cette fonctionnalité de cache.

3- Probleme du rendu de monnaie :

a. Exemple :

On suppose être dans un système monétaire dans lequel on ne trouve que des pièces de 1€, 3€ et 4€. Notre objectif est de créer un code qui permette de déterminer le nombre minimal de pièces à utiliser pour rendre une somme donnée. Par exemple :

- pour rendre une somme de 10 € :
- pour rendre une somme de 6 € :

b. ALGORITHME GLOUTON :

La fonction *glouton()* donnée ci-dessous est incomplète. Elle a en paramètres un nombre entier représentant une somme en € et une liste d'entiers représentant les pièces du système monétaire, triées par ordre croissant. Elle renvoie un message (string) donnant le nombre de pièces à rendre et une liste contenant les pièces à rendre. On donne 2 exemples d'exécution :

```
>>> glouton(10,[1,3,4])
('4 pièces à rendre', [4, 4, 1, 1])
>>> glouton(6,[1,3,4])
('3 pièces à rendre', [4, 1, 1])
```

⇨ Compléter ce code :

```
def glouton(somme , l) :
    rendu = []
    i = len(l)- 1
    s = somme
    while ..... and ..... :
        p = l[i]
        if p <= s :
            rendu.append(p)
            ..............
        else : ............

    message = str(len(rendu))+" pièces à rendre"
    return message , rendu
```

⇨Compléter le tableau ci-dessous qui donne l'état des variables au cours de l'exécution :

`glouton(6,[1,3,4])`

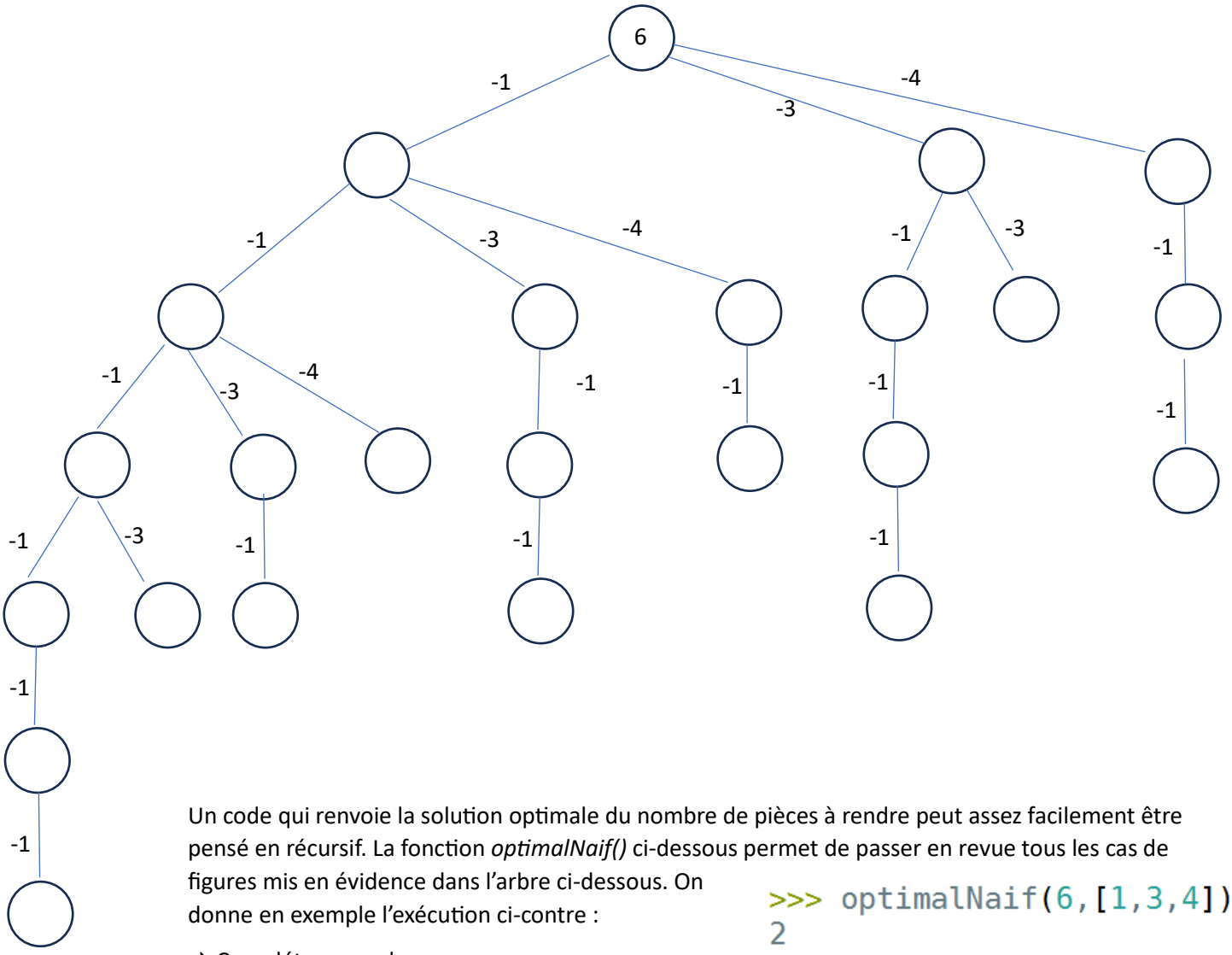
somme	*rendu*	*i*	*s*	*p*

Cet algorithme permet-il de trouver une solution optimale ?

L'heuristique gloutonne ne permet pas toujours de donner une solution optimale. Une *heuristique* par définition, ne donne pas forcément une solution exacte mais a pour but de l'approcher convenablement. L'intérêt de cette approche c'est sa simplicité : tant qu'il reste un montant à rendre on choisit la plus grande valeur de pièce disponible inférieure ou égale à la somme et on la retranche du montant à rendre.

c. SOLUTION OPTIMALE NAÏVE :

Pour être certain d'obtenir une solution optimale, il faut tester tous les cas de figure qui peuvent se présenter. Par exemple pour rendre une somme de 6 €, les différentes possibilités de rendus peuvent être visualisées par un arbre :



Un code qui renvoie la solution optimale du nombre de pièces à rendre peut assez facilement être pensé en récursif. La fonction *optimalNaif()* ci-dessous permet de passer en revue tous les cas de figures mis en évidence dans l'arbre ci-dessous. On donne en exemple l'exécution ci-contre :

```
>>> optimalNaif(6,[1,3,4])
2
```

⇨ Compléter ce code :

```
def optimalNaif(s,l) :
    if s == 0 : return ......   # Cas d'arrêt

    nbMinPieces = ......
    for p in l :
        if p <= s :
            nbPieces = 1 + optimalNaif( .........  ,  l)
            if nbPieces < nbMinPieces :
                nbMinPieces = ...........
    return nbMinPieces
```

⇨ Réaliser les exécutions ci-dessous et déterminer à peu près le temps d'exécution :

`>>> optimalNaif(10,[1,3,4])`	`>>> optimalNaif(30,[1,3,4])`	`>>> optimalNaif(50,[1,3,4])`
Temps :	Temps :	Temps :

⇨ Des 4 propositions qui suivent, laquelle semble la plus vraisemblable :

- Proposition 1 : La complexité de cette fonction est linéaire, en $\mathcal{O}(n)$
- Proposition 2 : La complexité de cette fonction est quadratique, en $\mathcal{O}(n^2)$
- Proposition 3 : La complexité de cette fonction est logarithmique, en $\mathcal{O}(\log_2(n))$
- Proposition 4 : La complexité de cette fonction est en $\mathcal{O}(\mathrm{n}\log_2(n))$
- Proposition 5 : La complexité de cette fonction est exponentielle, en $\mathcal{O}(k^n)$

d. SOLUTION OPTIMALE AVEC UNE APPROCHE PROGRAMMATION DYNAMIQUE :

Les temps de calcul deviennent rapidement très importants lorsque la somme à rendre augmente. On retrouve ici le même phénomène que celui constaté dans le calcul des termes de la suite de Fibonacci. Plusieurs mêmes calculs sont réalisés plusieurs fois et cela ralentit énormément l'exécution.

Dans le code ci-dessous, la fonction récursive est identique à la précédente, sauf qu'elle enregistre dans **un dictionnaire** nommé '*memo*' toute solution déjà calculée. Si le sous-problème a déjà sa solution stockée dans memo alors cette fonction la renvoie directement. Cette technique dans laquelle le dictionnaire memo agit comme un *cache mémoire*, est une **mémoïsation**. On donne en exemple l'exécution ci-contre :

```
>>> optimalDynamique(6,[1,3,4])
2
```

⇨ Compléter ce code :

```
def optimalDynamique(s,l):
    def avecMemo(s,l) :
        if s not in memo :
            nbMinPieces = ..............
            for p in l :
                if p <= s :
                    nbPieces = 1 + avecMemo(.......... , l)
                    if nbPieces < nbMinPieces :
                        nbMinPieces = nbPieces

            memo[s] = ......................
        return memo[s]

    memo = {}
    memo[0] = 0
    return  avecMemo(s, l)
```

⇨ Où se trouve le cas d'arrêt des récursions ?

⇨ Réaliser les exécutions ci-dessous et déterminer à peu près le temps d'exécution :

`>>> optimalDynamique(10,[1,3,4])` `3`	`>>> optimalDynamique(30,[1,3,4])` `8`	`>>> optimalDynamique(900,[1,3,4])` `225`
Temps :	Temps :	Temps :

⇨ Des 4 propositions qui suivent, laquelle semble la plus vraisemblable :

- Proposition 1 : La complexité de cette fonction est linéaire, en $\mathcal{O}(n)$
- Proposition 2 : La complexité de cette fonction est quadratique, en $\mathcal{O}(n^2)$
- Proposition 3 : La complexité de cette fonction est logarithmique, en $\mathcal{O}(\log_2(n))$
- Proposition 4 : La complexité de cette fonction est en $\mathcal{O}(\mathrm{n}\log_2(n))$
- Proposition 5 : La complexité de cette fonction est exponentielle, en $\mathcal{O}(k^n)$

e. SOLUTION ITERATIVE AVEC UNE APPROCHE PROGRAMMATION DYNAMIQUE :

Dans le code précédent, on a exploité la formule de récurrence nbPièces(s) = 1 + nbPièces(s-p) mise en évidence par la *décomposition en sous-problèmes,* pour construire une solution avec un algorithme récursif, ce qui correspond à une descente du *haut* (le problème initial) jusqu'au *bas* (les cas de base) . En anglais : *« Top-Down »*. Les calculs (sélection de la meilleure solution parmi toutes celles des sous-problèmes) sont cependant effectués dans la phase de remontée du *bas* vers le *haut*.

L'algorithme récursif est élégant mais cache la complexité des calculs puisque les sous-problèmes se chevauchent (contrairement aux algorithmes de type *Diviser Pour Régner*) et on a dû mettre en œuvre une technique de **mémoïsation** pour éviter les redondances de calcul et enregistrer les solutions des sous-problèmes dans une structure de données.

La fonction *optimalIteratif*() donnée ci-dessous, traduit dans un algorithme itératif la progression des calculs du *bas* vers le *haut*. En anglais : *« Bottom-Up »*. Pour construire la solution optimale d'une somme S, on parcourt tous les sous-problèmes d'une somme *s* allant de 0 à S. Pour chaque valeur de *s,* on calcule la solution optimale (nombre minimal de pièces) comme pour l'algorithme récursif en tirant partie du fait que les solutions des sous-problèmes plus petits ont déjà été calculées dans ici une **liste** nommée memo. On donne en exemple l'exécution ci-contre :

```
>>> optimalIteratif(6,[1,3,4])
2
```

⇨ Compléter ce code :

```
def optimalIteratif(somme,l):
    memo = [0 for i in range(somme+1)]
    for s in range(1,somme+1) :
        nbMinPieces = ..........
        for p in l :
            if p <= s :
                nbPieces = 1 + memo[...........]
                if nbPieces < nbMinPieces :
                    nbMinPieces = nbPieces
        ......................
    return memo[somme]
```

⇨ Réaliser les exécutions ci-dessous et déterminer à peu près le temps d'exécution :

`>>> optimalIteratif(10,[1,3,4])`	`>>> optimalIteratif(30,[1,3,4])` `8`	`>>> optimalIteratif(900,[1,3,4])` `225`
Temps :	Temps :	Temps :

⇨ Des 4 propositions qui suivent, laquelle semble la plus vraisemblable :

- Proposition 1 : La complexité de cette fonction est linéaire, en $\mathcal{O}(n)$
- Proposition 2 : La complexité de cette fonction est quadratique, en $\mathcal{O}(n^2)$
- Proposition 3 : La complexité de cette fonction est exponentielle, en $\mathcal{O}(k^n)$

f. SOLUTION ITERATIVE AVEC DETAIL DU RENDU DE MONNAIE :

On dispose désormais d'au moins deux algorithmes (un récursif et un itératif) permettant de déterminer l'optimum du nombre minimal de pièces à rendre. Dans ce paragraphe, on se propose d'améliorer le code itératif précédent en déterminant le détail du rendu de pièces.

Dans la fonction *optimalRendu()* donnée ci-dessous, on enregistre pour chaque somme s de la boucle représentant un sous-problème, la valeur p de la pièce qui ramène au sous-problème(s - p) donnant la

```
>>> optimalRendu(6,[1,3,4])
(2, [3, 3])
```

solution optimale pour s. On peut ainsi remplir un tableau nommée choix. Quand on est arrivé en *haut* (la somme du problème initial), on parcourt ce tableau choix à rebours pour reconstruire une liste de pièces optimale. On donne en exemple l'exécution ci-contre :

⇨ Compléter ce code :

```
def optimalRendu(somme,l):
    memo = [0 for i in range(somme+1)]
    choix = [0 for i in range(somme+1)]

    for s in range(1,somme+1) :
        nbMinPieces = s
        for p in l :
            if p <= s :
                nbPieces = 1 + memo[........]
                if nbPieces < nbMinPieces :
                    nbMinPieces = nbPieces
                    choix[s] = ............
        memo[s] = nbMinPieces

    rendu = [choix[somme]]
    s = somme - choix[somme]
    while ........
        ...................
        ...................

    return memo[somme] , rendu
```

⇨ Réaliser les exécutions ci-dessous et déterminer le contenu de la liste rendu :

`>>> optimalRendu(10,[1,3,4])`	`>>> optimalRendu(20,[1,3,4])`
rendu = [	rendu = [
`>>> optimalRendu(147,[1,2,5,10,20,50,100,200,500])` rendu = [	

g. CONCLUSION :

La **programmation dynamique** détermine une solution optimale d'un problème à partir des solutions optimales de sous-problèmes similaires (**sous-struture optimale**). Cela nous conduit naturellement à l'écriture d'un *algorithme récursif* de résolution.

Cependant, contrairement aux algorithmes **Diviser Pour Régner**, où les sous-problèmes sont indépendants, dans les problèmes de **programmation dynamique** les sous-problèmes peuvent se chevaucher.

Un algorithme *récursif* naïf va donc calculer plusieurs fois les solutions des mêmes sous-problèmes.

La **mémoïsation** permet d'éviter ces redondances de calcul : on améliore notablement la *complexité temporelle* (d'exponentielle à linéaire pour le rendu de monnaie) au prix d'une plus grande *complexité spatiale* :

- on enregistre chaque solution de sous-problème dans une structure de données memo (dictionnaire ou tableau qui permettent *l'accès en temps constant*)
- l'algorithme récursif vérifie d'abord si le sous-problème traité n'est pas déjà stocké dans memo avant de calculer sa solution et enregistre toute nouvelle solution dans memo

En **programmation dynamique**, un algorithme *récursif* procède de *haut* en *bas* en décomposant d'abord le problème et il doit être *mémoisé* pour être efficace. Mais, avec un algorithme *itératif*, on peut aussi effectuer les calculs de *bas* en *haut* en progressant des plus petits sous-problèmes jusqu'aux plus grands et en enregistrant toutes les solutions calculées dans un tableau.

Dans tous les cas, l'objectif de la **programmation dynamique** est d'éviter *la redondance des calculs* : on calcule toutes les solutions de sous-problèmes nécessaires mais une seule fois !