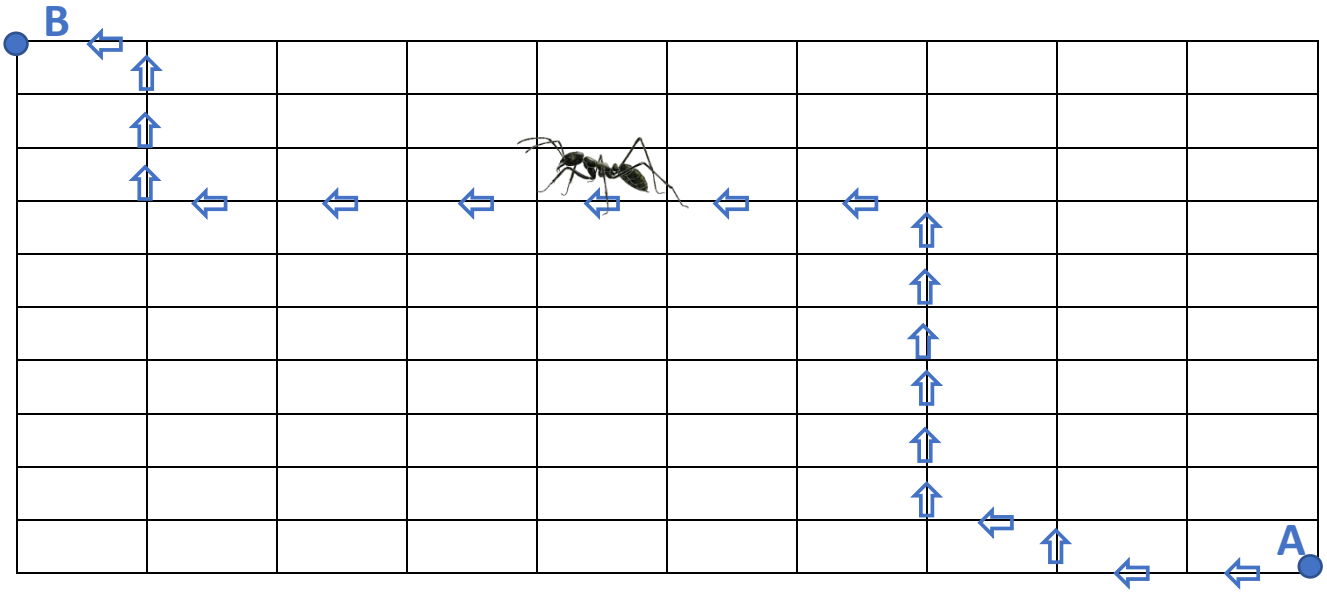
Chapitre 21 - Programmation dynamique

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 :

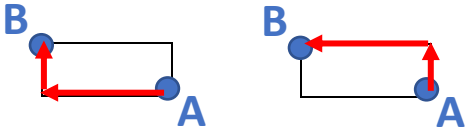
Alice pose le problème suivant à Basile. Elle dessine sur une feuille de papier une petite grille de N lignes et N colonnes. Par exemple, si $N = 10$, on obtient la figure suivante :



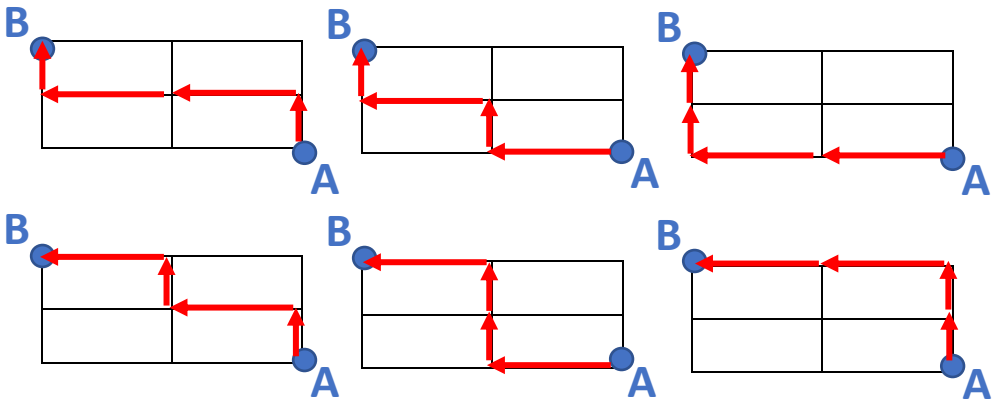
Une fourmi part du point **A** et veut aller sur le point **B**. Elle ne peut que suivre une ligne et se déplacer vers la gauche ou vers le haut, sans pouvoir revenir en arrière.

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

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

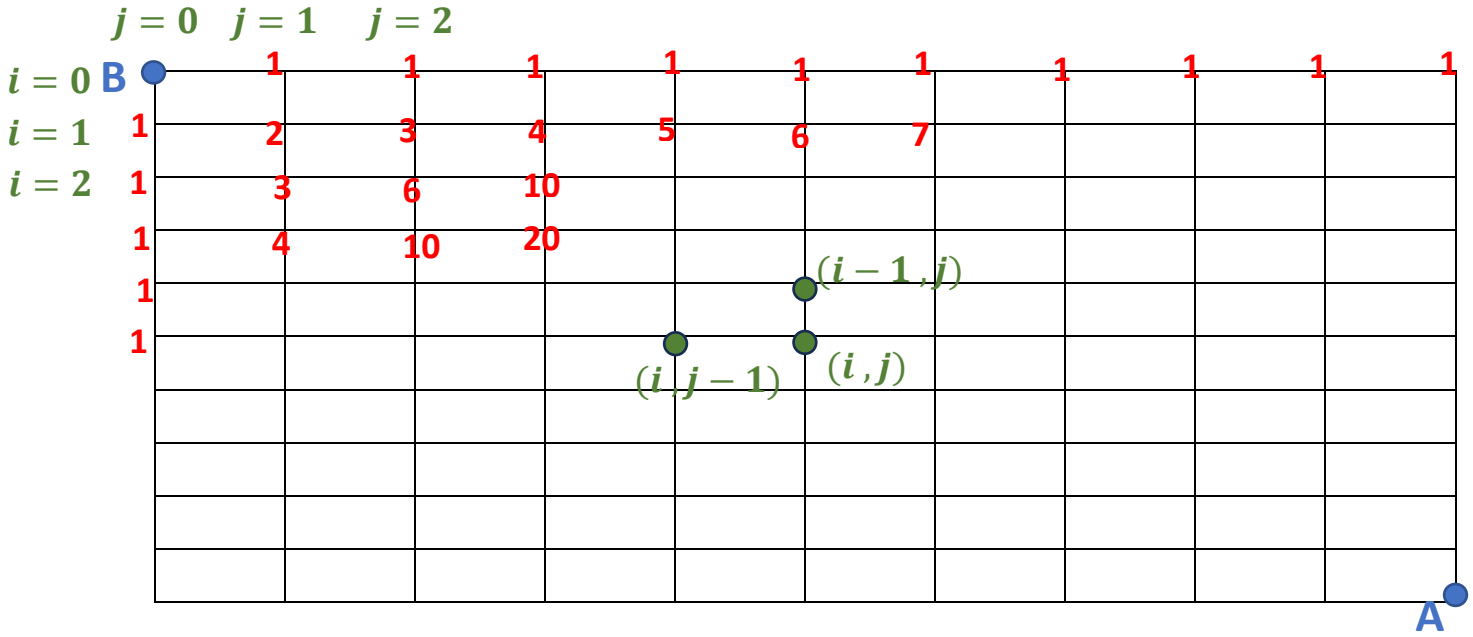


2- Nombre de chemins pour $N = 2$:



3- Nombre de chemins pour N quelconque :

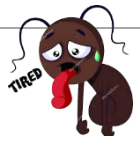
On se propose ici d'écrire un algorithme qui utilise le principe de la programmation dynamique. Cela consiste à résoudre le problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.



⇨ 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]
```

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.

Version récursive :

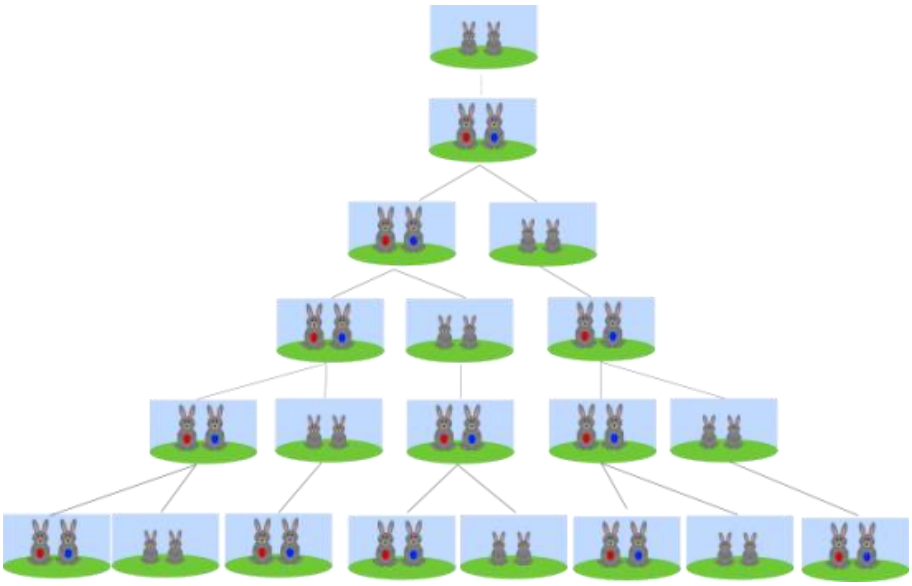
```
def fourmi_rec(i,j) :
    if i == 0 or j == 0 : return 1
    else :
        return fourmi_rec(i-1,j) + fourmi_rec(i,j-1)

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

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 :

$F_0 = 0$ et $F_1 = 1$.
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 » :

$F_{10} =$	$F_{20} =$	$F_{30} =$	$F_{40} =$	$F_{50} =$
Temps :	Temps :	Temps :	Temps :	Temps :


```
def fibNaif(n):
```

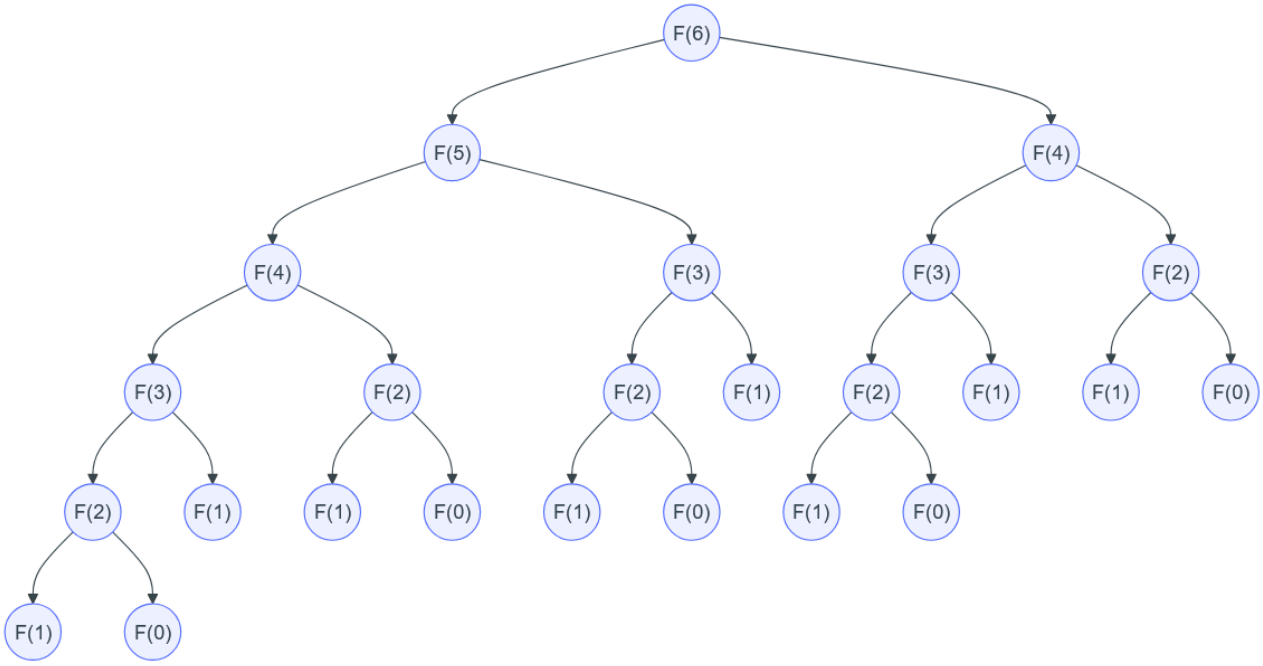
2- Algorithme récursif qui calcule un terme F_n en adoptant une méthode type programmation dynamique :

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 :

```
def fibDyn(n):
```

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

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

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

$F_{10} =$	$F_{50} =$	$F_{100} =$	$F_{500} =$	$F_{900} =$
Temps :	Temps :	Temps :	Temps :	Temps :