PROGRAMMATION DYNAMIQUE 1/3

- La programmation dynamique est une approche de résolution de problèmes qui consiste à décomposer un problème complexe en sous-problèmes plus simples (ce que permet aussi la récursivité ou le paradigme "diviser pour régner").
- La différence avec la récursivité est que chaque sous-problème ne sera résolu qu'une seule fois quand celui-ci se répète.
- Ce principe permet un gain de temps considérable.
- Le concept a été introduit au début des années 1950 par le mathématicien Richard Bellman(1920-1984).

1. Principes

Exercice 1 : Un algorithme naïf

- Dans la suite de Fibonacci, chaque terme à partir du troisième est égal à la somme des deux précedents. Les deux premiers termes sont égaux à 1.
- Cette suite commence donc ainsi : $1, 1, 2, 3, 5, 8, 13, \ldots$
- La fonction récursive ci-dessous calcule le terme de rang n de la suite de Fibonacci :

In [2]:
```
def f(n):
    if n<=1:
        return 1
    else :
        return f(n-1)+f(n-2)
```

1. Calculer les termes de rang $10, 20, 30, 40$.

- `f(10)` :
- `f(20)` :
- `f(30)` :
- `f(40)` :

In [3]:
```
#1.
f(1)
```

Out[3]: 1

2. Que peut-on dire de cet algorithme en terme de temps d'éxécution ?

-
-

3. Contruire l'arbre des appels pour `f(6)` .

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

4. Combien de fois est appelé `f(2)` ?

*

Remarques :

- La complexité de cet algorithme est exponentielle(ici en $O(2^n)$), c'est déplorable algorithmiquement. La raison de cette inefficacité se situe dans la nécéssité d'effectuer plusieurs fois le même calcul.
- Ainsi pour calculer `f(7)` , l'algorithme fait par exemple trois fois appel à `f(4)` , deux fois appel à `f(6)` , etc...

Exercice 2 : Vers une solution plus efficace

La clef d'une solution plus efficace serait de s'affranchir de la multiplicité des résolutions du même sous-problème. On améliore la complexité temporelle si, une fois calculé, on sauvegarde un résultat(dans un tableau par exemple). On peut ensuite réutiliser ce résultat si besoin.

1. Compléter la fonction récursive `f2(n,tab)` . Cette fonction prend en paramètres un entier `n` (le rang du terme à calculer) et un tableau `tab` qui contient les termes déjà calculés. On construit ce tableau de telle sorte que pour chaque indice `i` , `tab[i]` est le terme de rang `i` de la suite. Ainsi :
   - `tab[0]=1` , `tab[1]=1` , `tab[2]=2` , `tab[3]=3` , `tab[4]=5` , etc.
   - Pour calculer le terme de rang 40, on appelera `f2(40,[1,1])` .
1. En calculant différents termes, comparer la performance de cet algorithme avec le précédent.

In [4]:
```
#1. fonction
def f2(n,tab):
    if n<len(tab):
        pass
    else:
        pass
        return tab[n]
```

In [7]:
```
#2. tests
f2(1,[1,1])
```

Remarques:

- Cette approche du problème est dite "descendante" (top-down) : à partir du problème initial, on génére des sous-problèmes, on les résoud récursivement mais on mémorise les sous-problèmes déjà résolus(on parle de mémoïsation).
- L'idée est simple, mais le gain de temps est considérable.
- On note tout de même que la complexité en mémoire augmente puisqu'il faut stocker les résultats des sous-problèmes.

Exemple 3 : Programmation dynamique ascendante

On peut encore améliorer l'algorithme en se passant de la récursivité. On construit le tableau des termes à partir du premier jusqu'au rang souhaité.

1. Compléter la fonction `f3(n)` .
2. Tester quelques valeurs et comparer la performance de cet algorithme au précédent.

In [11]:
```
#1. fonction
def f3(n):
    tab=[1,1]
    i=2
    while i<=n:
        pass
        i=i+1

    return tab[n]
```

In [10]:
```
f3(1)
```

Out[10]: 1

Remarques :

- On y gagne encore en temps car il n' y a plus à identifier si telle ou telle solution a déjà été calculée. La complexité est celle de la construciton du tableau , linéaire en $O(n)$.
- On résoud ici le problème en partant des plus petits sous-problèmes. On parle d'approche ascendante (bottom-up).

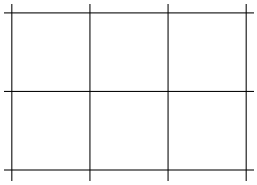
A retenir :

- La programmation dynamique est une technique pour améliorer l'éfficacité d'un algorithme en évitant les calculs redondants. Pour cela, on stocke dans un tableau les résultats intermédiaires du calcul afin de les réutiliser au lieu de les recalculer.
- La programmation dynamique n'est pas sans rapport avec la technique "diviser pour régner" présentée dans un autre document. En effet , on commence souvent par concevoir une décomposition récursive d'un problème pour se rendre compte ensuite que certains appels récursifs vont être effectués plusieurs fois. On utilise alors la programmation dynamque pour y remédier.

2. Exercices

Exercice 4 :

Alice pose le problème suivant à Basile. Elle dessine sur une feuille une petite grille 2×3 comme ci-dessous. Elle demande à Basile combien de chemins mènent du coin supérieur gauche au coin inférieur droit, en se deplaçant uniquement le long des traits horizontaux vers la droite et le long des traits verticaux vers le bas.Difficile de tous les énumérer de façon exhaustive...Mais Basile connaît les principes de la programmaiton dynamique. Il a alors l'idée de noter, à côté de chaque intersection le nombre total de chemins qui mènent au coin inférieur droit. Il commence par la fin et procède vers le haut et vers la gauche.



1. Ecrire sur la grille (ou la recopier ci-dessous)ce qu'a noté Basile et trouver le résultat.

-
-
-
-
-
-
-

2. Question pour les matheux : Combien y a-t-il de tels chemins possibles sur une grille 10×10 ?

Réponse :

-
-
-
-
-
-

Exercice 5 :

4	**1**	**1**	3
2	0	**2**	1
3	1	**5**	**1**

- On considère un tableau de nombres de n lignes et p colonnes. Les lignes sont numérotées de 0 à $n-1$ et les colonnes sont numérotées de 0 à $p-1$. La case en haut à gauche est repérée par $(0;0)$ et la case en bas à droite par $(n-1;p-1)$.
- On appelle chemin une succession de cases allant de la case $(0;0)$ à la case $(n-1;p-1)$, en n'autorisant que des déplacements case par case : soit vers la droite, soit vers le bas. On appelle somme d'un chemin la somme des entiers situés sur ce chemin.
- Par exemple, pour le tableau T ci-contre :
  - Un chemin est $(0;0), (0;1), (0;2), (1;2), (2;2), (2;3)$ (en gras sur le tableau).
  - La somme du chemin précédent est 14.
  - $(0;0), (0;2), (2;2), (2;3)$ n'est pas un chemin.
- La fonction ci-dessous permet de calculer la somme maximale pour tous les chemins possibles allant de la case (0; 0) à la case $(n-1;p-1)$. En particulier pour le tableau T donné en exemple, l'appel de `somme_max(T,2,3)` renvoie 16.

In [14]:
```
T=[[4,1,1,3],
   [2,0,2,1],
   [3,1,5,1]]

def somme_max(T, i, j):
    if i==0 and j==0:
        return T[0][0]
    else:
        if i==0:
            return T[i][j]+somme_max(T,i,j-1)
        elif j==0:
            return T[i][j]+somme_max(T,i-1,j)
        else:
            return T[i][j]+max(somme_max(T,i-1,j),somme_max(T,i,j-1))

print(somme_max(T,2,3))
```

```
16
```

1. Réecrire cette fonction ci-dessous à l'aide de la programmation dynamique. On construira un tableau contenant les solutions pour chaque case, sans utiliser la récursivité.

In [42]:
```
def somme_max_dyn(T):

    sol=[[T[0][0]]] #tableau(liste de listes python) contenant les solutions pour chaque case
    n=len(T) # nb lignes
    p=len(T[0]) #nb colonnes

    #construire les solutions de la première ligne
    pass

    #celles de la premiere colonne
    pass

    #reste du tableau des solutions
    for j in range(1,p):
        for i in range(1,n):
            pass


    return sol[n-1][p-1]

assert(somme_max_dyn(T)==16)
```

```
16
```

2. Ecrire la fonction `tableau(n,p)` qui génère un tableau à `n` lignes et `p` colonnes dont chaque case contient un entier naturel compris entre 0 et 9.

In [43]:
```
from random import randint
def tableau(n,p):
    pass
```

In [44]:
```
#tests
tableau(8,10)
```

Out[44]:
```
[[2, 4, 5, 0, 9, 5, 1, 0, 0, 0],
 [2, 9, 5, 5, 3, 8, 2, 1, 4, 0],
 [5, 3, 6, 1, 4, 7, 3, 7, 7, 8],
 [5, 5, 5, 3, 6, 6, 0, 5, 4, 8],
 [6, 4, 0, 4, 9, 5, 0, 8, 0, 0],
 [4, 5, 9, 5, 6, 6, 7, 9, 2, 7],
 [2, 7, 9, 7, 0, 6, 7, 2, 3, 9],
 [1, 8, 8, 3, 1, 2, 9, 9, 4, 4]]
```

3. Comparer l'efficacité de la fonction `somme_max()` et de la fonction `somme_max_dyn()` sur différents tableaux.

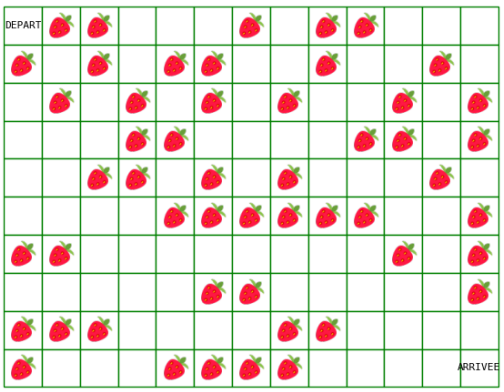
In [45]:
```
somme_max(tableau(13,13),12,12)
```

Out[45]:
```
143
```

In [46]:
```
somme_max_dyn(tableau(100,100))
```

Out[46]: 1358

Exercice 6 :



- Des fraises sont réparties sur un quadrillage rectangulaire comportant 10 x 13 cellules.
- Un robot chargé d'effectuer la cueillette part du coin supérieur gauche et doit déverser sa cueillette dans le coin inférieur droit.
- À chaque étape, le robot peut se déplacer d'une cellule soit vers la droite soit vers le bas (et ramasse uniquement les fraises situées sur les cellules qu'il traverse).

1. Modéliser ce champ de fraises à l'aide d'un tableau (liste de listes).
2. En utilisant la programmation dynamique, compléter la fonction `maxi_fraises(champ)` ci-dessous qui permet de déterminer combien de fraises au maximum le robot peut-il ramasser lors de son parcours. Elle prend en paramètre un champ de fraises modélisé par un tableau et renvoie le nombre maximal de fraises ramassables. On pourra s'appuyer sur la fonction de l'exercice 5.
3. En reprenant et en modifiant le code de la fonction de l'exercice 5, compléter la fonction `fraises(n,p)` ci-dessous qui génère un champ de fraises de n lignes et p colonnes, telles que les fraises sont réparties aléatoirement.
4. Tester l'algorithme précédent sur des champs différents.

In [47]:
```
#1.
```

In [50]:
```
#2
def maxi_fraises(champ):
    sol=[[champ[0][0]]]
    n=len(champ) #nb de lignes
    p=len(champ[0]) #nb de colonnes

    #construire les solutions de la première ligne
    pass

    #celles de la premiere colonne
    pass

    #reste du tableau des solutions
    pass

    return sol[n-1][p-1]

maxi_fraises(champ)
```

Out[50]: 15

In [59]:
```
#3
def fraises(n,p):
    pass

fraises(10,13)
```

Out[59]:
```
[[1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1],
 [0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0],
 [0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1],
 [0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0],
 [1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0],
 [0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0],
 [1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1],
 [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0],
 [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0]]
```

In [58]:
```
#4
maxi_fraises(fraises(20,20))
```

Out[58]: 30