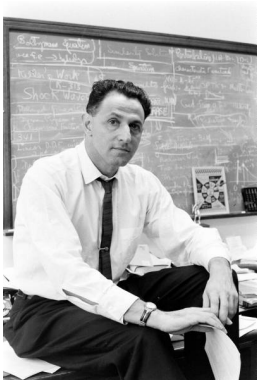
PROGRAMMATION DYNAMIQUE 1/3



- 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).

A. Principes

Exercice 1 : Un algorithme récursif

- Dans la suite de Fibonacci :
  - Les deux premiers termes sont égaux à 1.
  - Chaque terme à partir du troisième est égal à la somme des deux termes précédents.
- Cette suite commence donc ainsi : $1, 1, 2, 3, 5, 8, 13, \ldots$

1. Compléter la fonction récursive `f` ci-dessous. Elle prend en paramètre un entier $n \geq 0$ et renvoie le terme de rang n de la suite de Fibonacci.

Exemples de résultats attendus :

```
>>> f(0)     >>> f(1)     >>> f(6)
1            1            13
```

In [ ]:
```
def f(n):
    if n<=1:
        ...
    else :
        ...

assert f(0)==1
assert f(1)==1
assert f(6)==13
```

2. Calculons les termes de rang 10, 20 et 30.

In [ ]:
```
#2. Calculons
f(10)
```

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

3. A l'aide du module `time`, mesurons le temps de calcul des termes de rang 35, 36 et 37.

In [ ]:
```
from time import time
t0=time() # début du chronomètre
f(35) # calcul du terme de rang voulu
t1=time() # fin du chronomètre
print(t1-t0) # temps en secondes
```

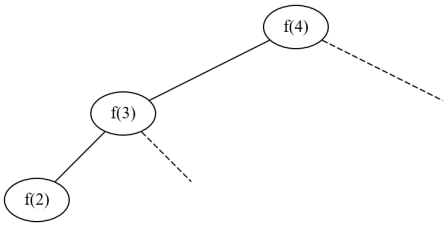
- `f(35)` :
- `f(36)` :
- `f(37)` :

4. Conclusion ?

-

5. Compléter l'arbre des appels pour `f(4)` . Combien de fois est appelé `f(2)` ?

- Nombres d'appels de `f(2)` :



6. Le nombre d'appels de la fonction `f` pour calculer `f(n)` correspond à la taille de l'arbre des appels.

a. A l'aide de l'arbre précédent, déterminer le nombre d'appels de la fonction `f` pour calculer `f(4)` , puis `f(3)` (le sous-arbre gauche de `f(4)`).

- Taille de l'arbre `f(3)` :
- Taille de l'arbre `f(4)` :

b. En déduire le nombre d'appels de la fonction `f` pour calculer `f(5)` , puis `f(6)` .

- Taille de l'arbre `f(5)` :
- Taille de l'arbre `f(6)` :

Conclusion :

- D'un rang au suivant, le nombre d'appels est multiplié par un facteur d'environ `1,5` .
- La complexité en temps de cet algorithme est dite exponentielle, c'est déplorable algorithmiquement (voir graphique ci-contre).
- La raison de cette inefficacité se situe dans la nécessité d'effectuer plusieurs fois le même calcul.
  - Par exemple pour calculer `f(7)` , l'algorithme fait trois fois appel à `f(4)` , deux fois appel à `f(6)` , ...
- Il nous faut trouver mieux !

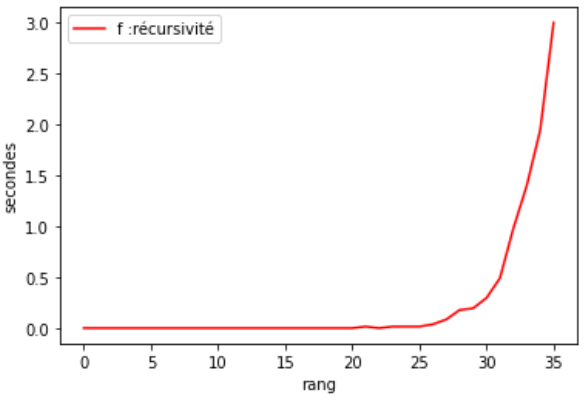


Figure : Mesures effectuées à l'aide du module time

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.

Utilisons un tableau `tab` qui contient les termes déjà calculés. Ce tableau est construit 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.

Descriptif de l'algorithme :

- Au début du programme, le tableau contient les 2 premiers termes de la suite de Fibonacci.
- Si le terme de rang n a déjà été calculé, c'est la valeur d'indice n du tableau, on la renvoie.
- Sinon on le calcule à l'aide des deux termes précédents, on l'ajoute au tableau et le renvoie.

1. A l'aide de la description ci-dessus, compléter la fonction récursive `f2(n)`. Cette fonction prend en paramètre un entier $n \geq 0$ et renvoie le terme de rang n de la suite de Fibonacci.

In [ ]:
```
#1. fonction f2 (version 1)
tab=...
def f2(n):
    if n < ... :
        return ...
    else:
        tab.append(      ...      )
        return tab[n]

assert f2(0)==1
assert f2(1)==1
assert f2(6)==13
```

2. Calculer les termes de rang 37 et 100.

In [ ]:
```
# tests v1
tab=[1,1]
f2(6)
```

- Terme de rang 37 :
- Terme de rang 100 :

3. Pour des raisons pratiques, le tableau peut-être passé en paramètre de la fonction. Compléter la 2ème version de `f2` ci-dessous :

In [ ]:
```
#1. fonction f2 (version 2)

def f2(n, tab):
    if n< ... :
        return ...
    else:
        tab.append(        ...        )
        return ...

assert f2(0,[1,1])==1
assert f2(1,[1,1])==1
assert f2(6,[1,1])==13
```

4. A l'aide du module `time`, mesurer le temps de calcul des termes de rang 37 puis 100.

In [ ]:
```
from time import time
t0=time() # début du chronomètre
f2(10,[1,1]) # calcul du terme de rang voulu
t1=time() # fin du chronomètre
print(t1-t0)
```

- Temps de calcul pour `f(37)` :
- Temps de calcul pour `f(100)` :

5. Que constate-t-on ?

-
-

⚠ Avant d'aller plus loin : Sauvegarder le travail. ⚠

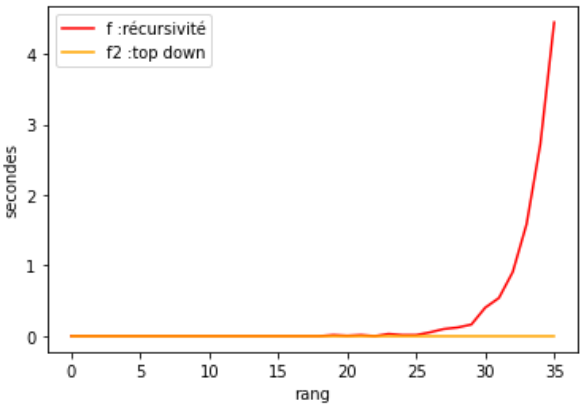
6. Chercher le plus grand rang pour lequel on peut calculer un terme de la suite :

In [ ]:
```
# test rang maximal
f2(6,[1,1])
```

Réponse:

Conclusion :

- 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 (voir graphique ci-contre).
- On note tout de même que la complexité en mémoire augmente puisqu'il faut stocker les résultats des sous-problèmes et que le nombre d'appels récursifs est limité.



Exercice 3 : Programmation dynamique ascendante

On peut encore améliorer l'algorithme en se passant de la récursivité. On construit le tableau des termes à partir des deux premiers jusqu'au rang souhaité qui est la dernière valeur du tableau.

1. A l'aide d'une boucle `for` ou une boucle `while` , compléter la fonction `f3(n)` . Elle prend en paramètre un entier $n \geq 0$ et renvoie le terme de rang n de la suite de Fibonacci.

In [ ]:
```
#1. fonction f3 (plus de récursivité)
def f3(n):
    tab=[1,1]
    # à compléter




    return tab[n]

assert f3(0)==1
assert f3(1)==1
assert f3(6)==13
```

2. Calculer `f3(5000)` , puis tester d'autres rang encore plus grands !

In [ ]:
```
#2. tests

#
```

3. Comparons le temps d'exécution de `f3` à celui de `f2` .Le module `timeit` permet effectuer des mesures de temps d'exécution de morceaux de programmes et donne des résultats plus pertinents que le module `time` .

A l'aide du code ci-dessous, comparer les temps de calcul du rang 800 par les fonctions `f2` , puis `f3` .

In [ ]:
```
import timeit
test2='f2(800,[1,1])' # test de f2 au rang 800
test3='f3(800)' # test de f3 au rang 800
t=timeit.Timer(test2,globals=globals()) # objet chronomètre
min(t.repeat(3,100)) # mesures effectués en secondes sur 3 échantillons de 100 tests, on garde le temps min
```

- Temps de calcul du terme de rang 800 avec `f2` :
- Temps de calcul du terme de rang 800 avec `f3` :

Conclusion :

- On y gagne encore car :
  - Il n'y a plus à identifier si telle ou telle solution a déjà été calculée (gain de temps).
  - Il n'y a plus d'appels récursifs (moins de mémoire occupée).
- La complexité en temps est celle de la construction du tableau, linéaire.
- On résoud ici le problème en partant des plus petits sous-problèmes. On parle d'approche ascendante (bottom-up).

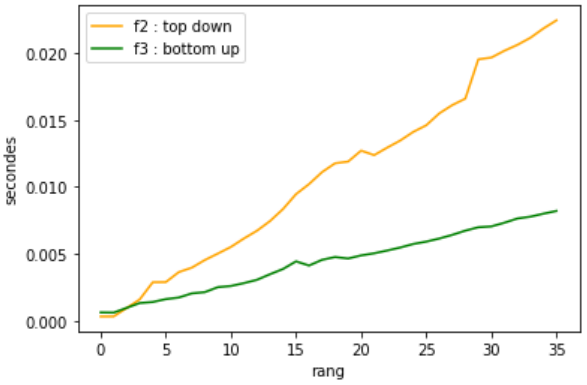


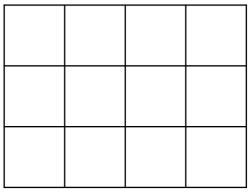
Figure : Mesures effectuées à l'aide du module timeit

A retenir :

- La programmation dynamique est une technique pour améliorer l'efficacité 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 dynamique pour y remédier.

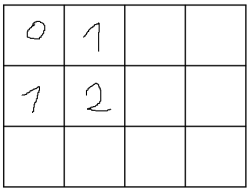
B. Exercices

Exercice 4 :



Alice pose le problème suivant à Bob. Elle dessine sur une feuille un quadrillage comme ci-contre. Elle demande à Bob combien de chemins mènent de la case en haut à gauche à la case en bas à droite, en se deplaçant uniquement vers la droite et vers le bas. Difficile de tous les énumérer de façon exhaustive...Mais Bob connaît les principes de la programmation dynamique. Il a alors l'idée de noter au crayon, dans chaque case le nombre total de chemins qui partent de la case en haut à gauche. Il commence par le début et procède vers la droite et vers le bas.

1. Compléter la grille commencée par Bob ci-dessous et trouver le résultat.



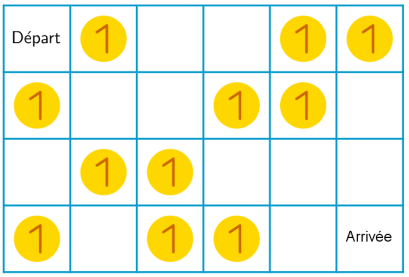
2. Question pour les matheux : Sans dessiner la grille, combien y a-t-il de tels chemins possibles sur une grille 4×6 ?

Réponse :

-
-

Exercice 5 :

Un robot est chargé de ramasser un maximum de pièces d'or réparties sur des quadrillages rectangulaires de 4×6 cellules (exemple ci-contre).



A chaque étape, il ne peut se déplacer que vers la droite ou vers le bas et ramasse uniquement les pièces situées sur les cellules qu'il traverse.

1. Combien de pièces au maximum le robot semble-t-il pouvoir ramasser sur le quadrillage ci-contre ?

Réponse:

2. En complétant le schéma ci-dessous, vérifier la réponse en utilisant une méthode similaire à celle de l'exercice précédent.

0	1	1	1	2	3
1	1				
1	2				
2	2				

3. On représente le quadrillage ci-dessus par un tableau de 4 tableaux contenant chacun 6 éléments, appelé `grille` . Les cellules contenant une pièce d'or sont représentées par des 1, les cellules vides par des 0.

In [ ]:
```
GRILLE=[[0,1,0,0,1,1],
        [1,0,0,1,1,0],
        [0,1,1,0,0,0],
        [1,0,1,1,0,0]]
```

a. Quelle est la valeur de `GRILLE[2][1]` ?

Réponse:

b. La case "arrivée" ne contient pas de pièce. Quelle instruction permet de le vérifier ?

In [ ]:
```
# Réponse :

#
```

4. La fonction récursive `maxi_pieces` ci-dessous prend en paramètres un tableau `grille` et deux entiers `i` et `j`. Elle renvoie le nombre maximal de pièces que le robot peut ramasser la première case en haut à gauche jusqu'à la case d'indices `i` et `j`.

Exemples :

- L'appel de `maxi_pieces(GRILLE,3,5)` renvoie `5`.
- L'appel de `maxi_pieces(GRILLE,1,3)` renvoie `2`.

En complétant les commentaires `#...`, expliquer ce que fait cette fonction.

In [ ]:
```
def maxi_pieces(grille, i, j):
    if i==0 and j==0:
        #...
        return grille[0][0]
    else:
        if i==0:
            #...
            return grille[i][j] + maxi_pieces(grille,i,j-1)
        elif j==0:
            #...
            return grille[i][j] + maxi_pieces(grille,i-1,j)
        else:
            #...
            return grille[i][j] + max(maxi_pieces(grille,i-1,j),maxi_pieces(grille,i,j-1))

print(maxi_pieces(GRILLE,3,5))
```

3. La fonction ci-dessous génère une grille de n lignes sur p colonnes contenant un nombre aléatoire de pièces.

In [ ]:
```
from random import *
def pieces(n,p):
    '''renvoie un nombre aléatoire de pièces
     sur un quadrillage de n lignes et p colonnes'''
    return [[1 if random() < 0.5 else 0 for _ in range(p) ] for _ in range(n)]
```

a. Afficher une grille de 4 lignes sur 6 colonnes différente de la grille des questions précédentes, puis tester la fonction `maxi_pièces` sur différentes grilles de ces dimensions.

In [ ]:
```
#une grille de 4x6


#
```

In [ ]:
```
#tests maxi_pieces

#
```

⚠ Avant d'aller plus loin : Sauvegarder le travail. ⚠

b. Tester la fonction `maxi_pièces` sur des grilles de différentes dimensions. Que constate-t-on ?

In [ ]:
```
GRILLE=pieces(4,6) #modifier les dimensions de la grille
N=len(GRILLE)-1
P=len(GRILLE[0])-1
maxi_pieces(GRILLE,N,P)
```

Réponse :

4. Faisons donc de la programmation dynamique ascendante (bottom up) !

a. Compléter la fonction `maxi_pieces_dyn(grille)` . Cette fonction prend en paramètre une grille et renvoie le nombre maximal de pièces que le robot peut ramasser. Elle utilise la programmation dynamique ascendante (bottom-up). On construit le tableau des solutions en partant des cas les plus simples. Il n'y a pas de récursivité.

In [ ]:
```
def maxi_pieces_dyn(grille):

    n=len(grille) # nombre de lignes de la grille
    p=len(grille[0]) # nombre de colonnes de la grille
    solutions=[[0 for _ in range(p) ] for _ in range(n)] # le tableau des solutions initialisé avec des 0
    solutions[0][0]=grille[0][0] #la solution de la première case

    # Solutions de la première ligne
    for j in range(1,p):
        ...

    #Solutions de la premiere colonne
    for i in range(1,n):
        ...

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

    # on renvoie la valeur contenue dans la case "inférieure droite" du tableau
    return ...
```

In [ ]:
```
#test
GRILLE=pieces(4,6)
maxi_pieces_dyn(GRILLE)
```

b. Tester cette fonction sur de grandes grilles et constater l'efficacité de cette programmation.

In [ ]:
```
#tests grandes grilles



#
```

Exercice 6 :

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 [ ]:
```
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))
```

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









    return

print(somme_max_dyn(T))
```

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 [ ]:
```
from random import randint



#
```

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

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

In [ ]:
```
#tests somme_max

#
```

In [ ]:
```
#tests_somme_max_dyn

#
```

FIN