

Programmation dynamique

I. Un premier exemple

Activité n°1:

Partie A

La suite de Fibonacci est la suite d'entiers (u_n) définie par:

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \quad \text{pour } n \geq 2 \end{cases}$$

1. Compléter le tableau suivant en calculant les valeurs de la suite "à la main".

n	0	1	2	3	4	5	6
u_n	0	1					

2. Compléter la fonction récursive u qui permet de calculer le terme de rang n (avec $n \geq 0$) de la suite.
3. Compléter le tableau suivant en exécutant la fonction u avec $n = 3$ puis $n = 4$ et $n = 5$.

```
def u(n):  
    print("Calcul de u_" + str(n) )  
    if n == 0:  
        return 0  
    elif n == 1:  
        return .....  
    else:  
        return .....
```

Calcul de $u(n)$	Nombre de fois où le calcul de $u(n)$ est effectué						nombre total d'appels t_n
	$u(5)$	$u(4)$	$u(3)$	$u(2)$	$u(1)$	$u(0)$	
$u(0)$						1	1
$u(1)$					1		1
$u(2)$				1	1	1	3
$u(3)$							
$u(4)$							
$u(5)$							

4. On note (t_n) le nombre total d'appels à la fonction u pour calculer $u(n)$. On a donc $t_0 = 1$, $t_1 = 1$, $t_2 = 3$...

- a. Donner les valeurs de t_6 et t_7 sans utiliser le programme.
- b. Donner une relation entre t_n , t_{n-1} et t_{n-2} .
- c. On donne les temps de calcul (en sec) de $u(n)$ sur une machine A pour différentes valeurs de n .
Estimer le temps pour calculer $u(38)$ sur cette machine.

```
n= 10 temps= 4.830000000000112e-05  
n= 15 temps= 0.0005214000000000052  
n= 20 temps= 0.0056917000000000077  
n= 25 temps= 0.043266499999999875  
n= 30 temps= 0.447158200000000006  
n= 34 temps= 3.0418830999999997  
n= 35 temps= 4.9533066  
n= 36 temps= 7.9799861000000001  
n= 37 temps= 12.9505608000000002
```

5. Expliquer pourquoi la méthode récursive n'est pas très efficace pour le calcul de u_n .

Partie B

On a remarqué que lors des appels récursifs, beaucoup d'appels étaient réalisés plusieurs fois.

Une nouvelle approche consiste à sauvegarder les résultats intermédiaires dans un tableau pour éviter de calculer deux fois la même chose.

Cette méthode est appelée programmation fonctionnelle.

On souhaite calculer $u(5)$.

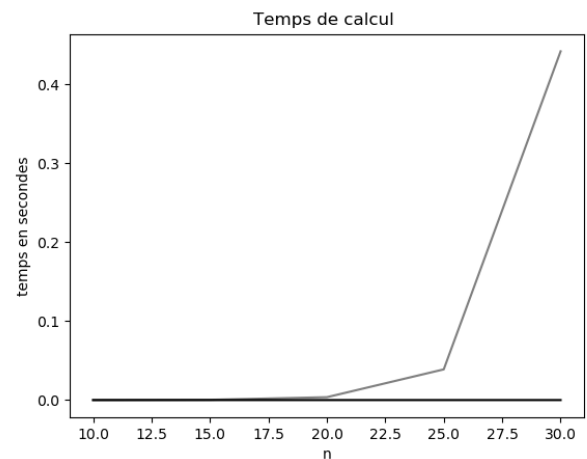
On va créer un tableau (de longueur 6) et enregistrer les valeurs de $u(0), u(1), \dots, u(5)$.

1. Compléter le tableau donnant l'évolution de la valeur de t au cours de l'exécution de l'appel `fibonacci(5)`.

k	t
////////	[0, 1, 0, 0, 0, 0]
2	
3	
4	
5	

```
def fibonacci(n):  
    t = [0]*(n+1)  
    t[0] = 0  
    t[1] = 1  
  
    for k in range(2,n+1):  
        t[k] = t[k-1]+t[k-2]  
  
    return t[n]
```

2. On a tracé les courbes représentant les temps de calcul en fonction de n pour les deux méthodes.
 - a. Identifier chacune des courbes;
 - b. Compléter:
La programmation fonctionnelle possède une efficacité:
 - linéaire (proportionnelle à n);
 - quadratique (proportionnelle à n^2);
 - exponentielle.
 - c. Même question pour la méthode récursive.



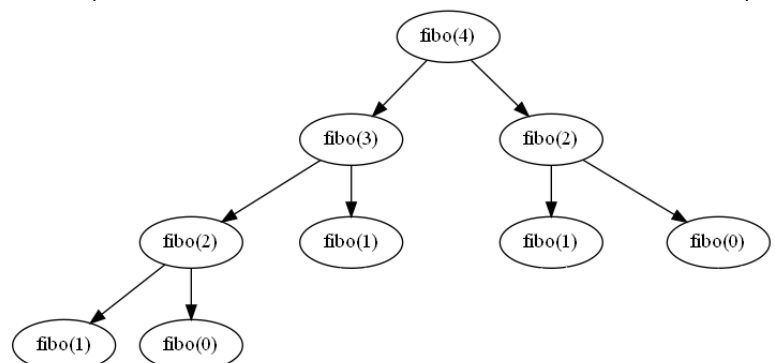
Partie C

On peut adapter le principe de programmation fonctionnelle (garder les informations lors des calculs intermédiaires) en utilisant une fonction récursive.

On place alors les calculs déjà effectués dans un dictionnaire. C'est la méthode de mémoïsation.

1. Donner la valeur de la variable `dico` après l'exécution de `fibonacci(2)`.
2. On utilise le programme ci-contre en réalisant l'appel `fibonacci(4)`. Surligner :
 - en rouge les nœuds de l'arbre ci-dessous qui doivent être calculés (autrement dit les appels qui ajoutent une nouvelle clé dans le dictionnaire);
 - en vert les nœuds de l'arbre ci-dessous qui sont déjà calculés.
 - barrer les nœuds inutiles dans le schéma.

```
dico = {}  
dico[0] = 0  
dico[1] = 1  
  
def fibo(n):  
    if n in dico:  
        return dico[n]  
    else:  
        dico[n] = fibo(n-1)+fibo(n-2)  
        return dico[n]
```



Exercice n°1:

Partie A: Rappels

- Relier chaque tableau à sa représentation:

t1 = [[0]*2 for k in range(5)] • • [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]

t2 = [[0]*5 for k in range(2)] • •

t3 = [[0 for i in range(5)] for k in range(2)] • • [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]


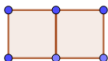
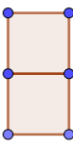
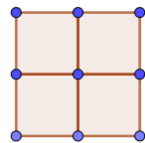
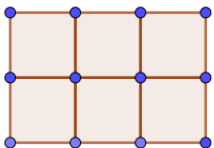
t4 = [[0 for i in range(2)] for k in range(5)] • •

Partie B:

On dessine une grille de hauteur h et de largeur l constituée de carrés.

On cherche à déterminer le nombre de chemins possibles pour aller **du coin en haut à gauche jusqu'au coin en bas à droite, en se déplaçant uniquement vers le bas ou vers la droite** (le long des côtés des carrés).

- Compléter le tableau suivant:

	$h = 1 ; l = 1$	$h = 1 ; l = 2$	$h = 2 ; l = 1$	$h = 2 ; l = 2$	$h = 2 ; l = 3$
					
nombre de chemins possibles					

- On cherche à définir une fonction `chemins(h , l)` qui renvoie le nombre de chemins possibles pour une grille de taille $h \times l$.

a. Compléter la formule suivante (en supposant $h > 1$ et $l > 1$) en justifiant:

$$\text{chemins}(h , l) = \text{chemins}(\dots , \dots) + \text{chemins}(\dots , \dots)$$

b. En déduire la valeur de `chemins(3 , 3)`.

- Utiliser la programmation dynamique pour programmer la fonction `chemins(h , l)`.

Aide: Il faut enregistrer les informations dans un tableau `t` à 2 dimensions.

`t[1][2] = 3` signifie que pour une grille de taille 1×2 , le nombre de chemins vaut 3.

Exercice n°2:

Le triangle de Pascal (nommé ainsi en l'honneur du Mathématicien Blaise Pascal) est une présentation des coefficients binomiaux sous forme d'un triangle définie ainsi de manière récursive:

$$C(n, k) = \begin{cases} 1 & \text{si } k = 0 \text{ ou } k = n \\ C(n-1, k-1) + C(n-1, k) & \text{sinon} \end{cases}$$

- Donner les valeurs de $C(1,0)$, $C(1,1)$, $C(2,0)$ et $C(2,2)$. Placer ces valeurs dans le tableau ci-dessous.
- Calculer la valeur de $C(2,1)$ et placer la valeur dans le tableau.
- Quelles valeurs du tableau a-t-on besoin pour calculer $C(4,2)$?
- Compléter le tableau.

k	0	1	2	3	4
n					
0	1	////////	////////	////////	////////
1			////////	////////	////////
2				////////	////////
3					////////
4			????		

- Ecrire une fonction récursive $c(n, p)$ qui renvoie la valeur de $c(n, p)$.
- Modifier votre fonction en utilisant le principe de mémorisation (vu dans la partie C de l'activité n°1).

Aide: Les clés du dictionnaire peuvent être des tuples de la forme (n, k) .

Exemple:

$d = \{(\emptyset, \emptyset): 1, (\emptyset, 2): 1, (\emptyset, 1): 1, (1, 1): 1, (1, 2): 2, \dots\}$

II. Exemple du rendu de monnaie

Activité n°2:

Dans un pays imaginaire, le système monétaire ne comporte que trois pièces différentes.

Des pièces de 1 €, 4€ et 5€.

On considère le problème de rendu de monnaie:

On souhaite rendre un montant m € avec le moins de pièces possibles.

- Rappeler le principe de l'algorithme glouton avec un montant de 12 €. Justifier que cet algorithme n'est pas optimal avec ce système monétaire.
- Compléter le tableau suivant:

montant m	nombre de pièces de 1€	nombre de pièces de 4€	nombre de pièces de 5€	nombre minimal de pièces rendues
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

On peut alors définir un tableau en Python avec les informations sur le nombre minimal de pièces rendues pour des montants compris entre 0 et 10.

$np = [0, 1, 2, 3, 1, \dots, 2]$

- Donner la valeur de np .

On souhaite déterminer le nombre minimal de pièces à rendre pour un montant de 11 € en utilisant le tableau précédent.

Nous avons trois possibilités pour la première pièce rendue:

- Choix n°1 : une pièce de 1 € ; Dans ce cas, il reste 10 € ;
- Choix n°2 : une pièce de 4 € ; Dans ce cas, il reste 7 € ;
- Choix n°3 : une pièce de 5 € ; Dans ce cas, il reste 6 € ;

	nombre de pièces
6€	2
7€	3
10€	2

Quel est le meilleur choix? Pour rendre le moins de pièces possibles, il faut regarder le nombre de pièces rendues dans le tableau précédent pour 6€, 7€ et 10 €.

Ici, on choisit entre le choix n°1 et le choix n°3.

La solution optimale pour rendre 11 € est donc de 3 pièces (Choix n°1: 1 + 5 + 5 ou Choix n°3: 5 + 1 + 5).

On peut alors mettre à jour le tableau np :

np = [0, 1, 2, 3, 1,..., 2, 3]
(indice 0 1 2 3 4 10 11)

4. On utilise Python pour effectuer l'étape précédente.

On note E le tableau avec les différentes valeurs des pièces disponibles.

Donner la valeur des variables E_11, liste_11 et np à la fin du programme ci-contre.

```
np = [0, 1, 2, 3, 1,..., 2] # indice de 0 à 10
E = [1,4,5]
E_11 = [ 11 - p for p in E]
liste_11 = [ np[m] for m in E_11]
np.append( 1 + min (liste_11) )
```

5. Compléter le programme ci-contre afin qu'il calcule le nombre minimal de pièces à rendre pour des montants compris entre 0 € et 30 €.

On ajoutera un affichage semblable à celui proposé ci-contre.

6. On suppose maintenant que les pièces disponibles valent 1 €, 4 € et 6€.

Programmer une fonction rendu_monnaie(m) qui, pour un montant de m € donné en paramètre, renvoie le nombre minimal de pièces nécessaires pour réaliser la somme m.

```
montant = 0; nbre de pièces 0
montant = 1; nbre de pièces 1
montant = 2; nbre de pièces 2
montant = 3; nbre de pièces 3
montant = 4; nbre de pièces 1
montant = 5; nbre de pièces 1
```

Aide: Il n'est pas nécessaire de calculer à la main les valeurs pour m compris entre 0 et 10.

7. Pour aller plus loin: Même question avec le système monétaire européen (1 €, 2€, 5€, 10€, 20€, 50€, 100€, 200€, 500€).

Exercice n°3: Alignement de séquences

On cherche à aligner deux chaînes de caractères. On utilise pour cela les règles d'alignement suivantes:

- on conserve l'ordre des lettres dans les deux mots (et on utilise toutes les lettres des mots);
- on peut insérer des trous dans l'une ou l'autre chaîne (un trou est représenté par le symbole -);
- on ne peut pas aligner deux trous.

Exemple avec les mots GENOME et ENORME:

alignement n°1							alignement n°2							alignement n°3							alignement n°4						
G	E	N	O	M	E		G	E	N	O	M	E	—	G	E	N	O	—	M	E	G	E	N	O	—	M	E
E	N	O	R	M	E		—	E	N	O	R	M	E	E	N	O	R	M	E	—	—	E	N	O	R	M	E

Afin d'évaluer la "proximité" entre deux mots, on va alors chercher le meilleur alignement de séquences en utilisant les règles suivantes:

- deux lettres identiques alignées rapportent 1 point;
- deux lettres différentes alignées enlèvent 1 point (ce qui inclut l'alignement avec le caractère -)

Ainsi, les notes pour les quatre alignements ci-dessus sont:

alignement n°1							alignement n°2							alignement n°3							alignement n°4							
<i>G</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>M</i>	<i>E</i>		<i>G</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>M</i>	<i>E</i>	—	<i>G</i>	<i>E</i>	<i>N</i>	<i>O</i>	—	<i>M</i>	<i>E</i>	<i>G</i>	<i>E</i>	<i>N</i>	<i>O</i>	—	<i>M</i>	<i>E</i>	
<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>	<i>M</i>	<i>E</i>		—	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>	<i>M</i>	<i>E</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>	<i>M</i>	<i>E</i>	—	—	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>	<i>M</i>	<i>E</i>	
−1	−1	−1	−1	1	1		−1	1	1	1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	1	1	1	−1	1	1
<i>score</i> = $4 \times (−1) + 2 = −2$							<i>score</i> = −1							<i>score</i> = −6							<i>score</i> = 3							

1. On admet que le score maximal entre les mots "astucieux" et "studieux" vaut 5. Trouver l'alignement qui réalise ce score.
2. On cherche à définir en Python une fonction `alignement(s1 ,s2)` qui renvoie le score maximal des alignements entre les chaînes `s1` et `s2`.

Donner les valeurs renvoyées par les appels suivants:

- `alignement("genome" ,"enorme")`
- `alignement("sept" ,"car")`
- `alignement("sept" ,"art")`
- `alignement("arbre" ,"")`
- `alignement("" ,"electricite")`
- `alignement("" ,"")`

3. On va utiliser la programmation dynamique pour résoudre ce problème.

Pour trouver le meilleur alignement entre les chaînes `s1` et `s2`, on peut choisir l'un des trois cas ci-dessous:

- Cas n°1: On aligne les dernières lettres des deux mots;

Exemple: l'alignement n°1.

- Cas n°2: On place un trou à la fin du premier mot;

Exemple: l'alignement n°2

- Cas n°3: On place un trou à la fin du deuxième mot;

Exemple: l'alignement n°3.

On appliquant l'une de ces méthodes, on se ramène à l'alignement de mots plus courts:

Exemple avec les mots `GENOME` et `ENORME` (6 lettres et 6 lettres):

- Cas n°1: Il reste à aligner `GENOM` et `ENORM` (5 lettres et 5 lettres);
- Cas n°2: Il reste à aligner `GENOME` et `ENORM` (6 lettres et 5 lettres);
- Cas n°3: Il reste à aligner `GENOM` et `ENORME` (5 lettres et 6 lettres);

On a donc ramené le problème à un problème plus simple.

4. Compléter le programme ci-dessous qui enregistre progressivement les meilleurs scores des alignements des mots `s1[0:i]` et `s2[0:j]` (dans un tableau à 2 dimensions `t`), en commençant par les mots "les plus petits" jusqu'au mot `s1` et `s2`.
5. Programmer puis tester cette fonction. La trace de l'appel `alignement("enorme", "genome")` est affichée dans le cadre ci-contre.

i	j	s1[0:i]	s2[0:j]	score
0	0			0
1	0	e		-1
2	0	en		-2
3	0	eno		-3
4	0	enor		-4
5	0	enorm		-5
6	0	enorme		-6
0	0			0
0	1		g	-1
0	2		ge	-2
0	3		gen	-3
0	4		geno	-4
0	5		genom	-5
0	6		genome	-6
1	1	e	g	-1
1	2	e	ge	0
1	3	e	gen	-1
1	4	e	geno	-2
1	5	e	genom	-3
1	6	e	genome	-4
2	1	en	g	-2
2	2	en	ge	-1
2	3	en	gen	1
2	4	en	geno	0
2	5	en	genom	-1
2	6	en	genome	-2
3	1	eno	g	-3
3	2	eno	ge	-2
3	3	eno	gen	0
3	4	eno	geno	2
3	5	eno	genom	1
3	6	eno	genome	0
4	1	enor	g	-4
4	2	enor	ge	-3
4	3	enor	gen	-1
4	4	enor	geno	1
4	5	enor	genom	1
4	6	enor	genome	0
5	1	enorm	g	-5
5	2	enorm	ge	-4
5	3	enorm	gen	-2
5	4	enorm	geno	0
5	5	enorm	genom	2
5	6	enorm	genome	1
6	1	enorme	g	-6
6	2	enorme	ge	-4
6	3	enorme	gen	-3
6	4	enorme	geno	-1
6	5	enorme	genom	1
6	6	enorme	genome	3

```
def alignement( s1, s2 ):
    t = [ [0]*(len(s2)+1) for k in range((len(s1)+1)) ]
    for i in range(0 , len(s1)+1 ):
        t[i][0] = .....
    for j in range(0 , len(s2)+1 ):
        t[0][j] = .....
    for i in range(1 , len(s1)+1 ):
        for j in range(1 , len(s2)+1 ):
            fin1 = s1[i-1]
            fin2 = s2[j-1]
            #choix n°1 : alignement des dernières lettres
            c1 = t[i-1][j-1]
            #choix n°2: on place un - à la fin du premier mot
            c2 = .....
            #choix n°3: on place un - à la fin du deuxième mot
            c3 = .....

            if fin1 == fin2:
                t[i][j] = .....
            else:
                t[i][j] = max( c1 - 1 , c2 - 1 , c3 - 1 )
    return t[len(s1)][len(s2)]
```