III

Dictionnaires 3 : utilisation en programmation dynamique

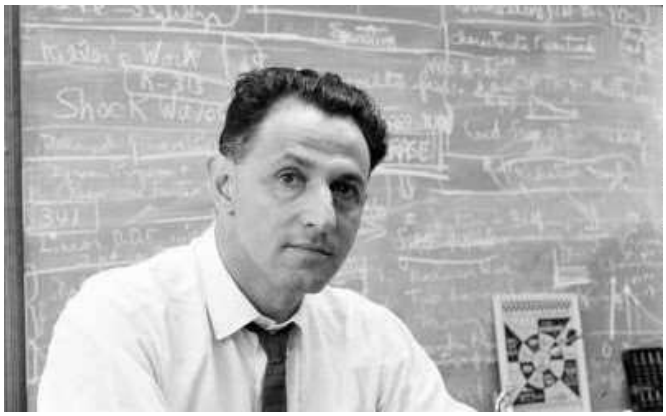


FIGURE III.1 – Richard Bellman, inventeur de la programmation dynamique (1920-1984)

Richard Bellman est né le 26 août 1920 à New York. À la fin de ses études universitaires à Baltimore, il est d'abord instructeur des armées avant d'être affecté au projet Manhattan entre 1944 et 1946. Il prépare ensuite une thèse sur les équations différentielles à Princeton sous la direction de Lefschetz et commence une carrière académique. Attiré par la théorie des nombres, il est aussi séduit par les défis mathématiques posés par les applications. Il décide de rejoindre la RAND corporation en 1952. C'est à cette période qu'il formule le concept de programmation dynamique, qui aura des répercussions scientifiques et techniques considérables. Il reprend sa carrière universitaire en 1965 à la University of Southern California. (Source : http ://www.breves-de-maths.fr/*)*

PLAN DU CHAPITRE

I Un nouveau paradigme : la programmation dynamique

I.1 Limite de la récursivité : retour sur la suite de Fibonacci

Parmi les exemples souvent cités dans les cours sur la récursivité, la suite de Fibonacci occupe une bonne place.

Rappelons d'abord sa définition ; le $n^{\text{ième}}$ terme de la suite de premiers termes $Fib(0) = 0$ et $Fib(1) = 1$ s'écrit :

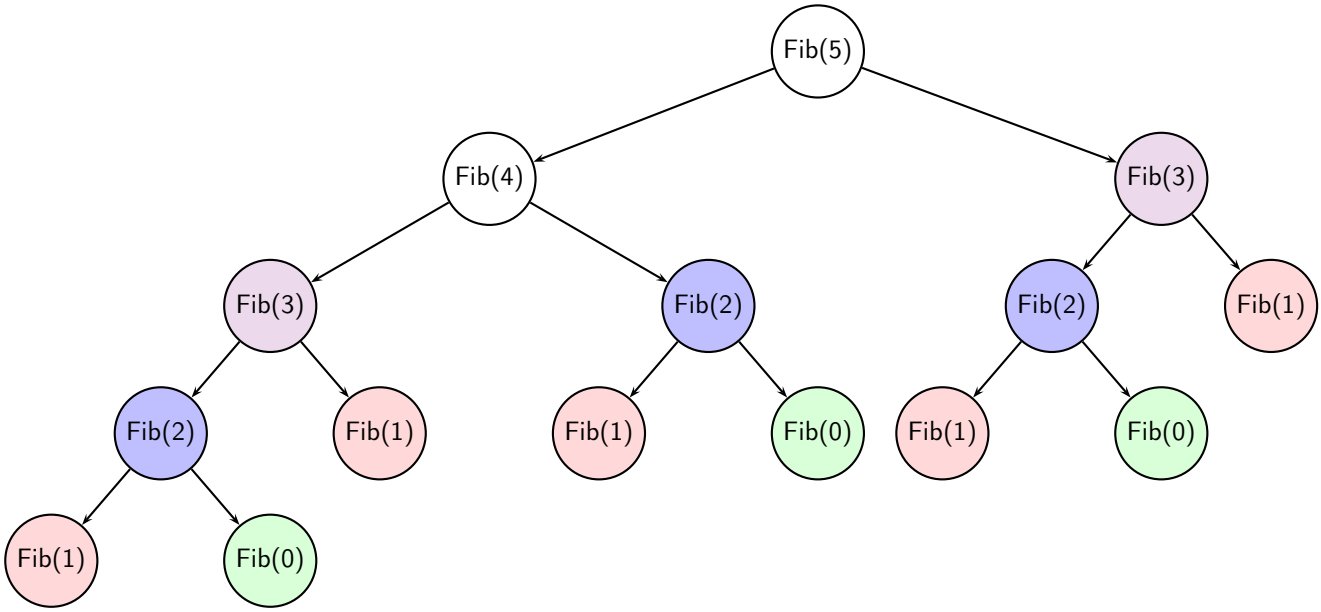
$$Fib(n) = Fib(n-1) + Fib(n-2)$$

et sa programmation récursive est :

Listing III.1 – Fonction récursive de calcul du nième terme de la suite de Fibonacci

```
1 def Fiborec(n):
2   if n<=1:
3     return n
4   else:
5     return Fiborec(n-1)+Fiborec(n-2)
```

Malheureusement cette fonction est peu performante, et permet d'illustrer que la récursivité n'est pas toujours une approche judicieuse. En effet, l'arbre des appels pour le calcul de Fib(5) montre que de nombreux termes sont recalculés plusieurs fois :



Dans le détail, le calcul de `Fib(5)` nécessite d'évaluer 2 fois `Fib(3)`, 3 fois `Fib(2)`, 5 fois `Fib(1)`, et 3 fois `Fib(0)`. On imagine sans peine que le calcul de `Fib(100)`, par exemple, va conduire à une véritable «explosion» de la complexité temporelle !

Evaluation de la complexité temporelle :
Posons $C(n)$ le nombre de calculs effectués pour l'évaluation de `Fib(n)` ; $C(n)$ vérifie la récurrence suivante :

$$\forall n \geq 2,\ C(n) = C(n-1) + 1 + C(n-2) + 1 \qquad \text{(2 pour le test de } n \text{ et l'addition de Fib(n-1) et Fib(n-2))}$$

Posons la suite $U(n) = C(n) + 2$ (puisque -2 est le point fixe de cette récurrence) ; il vient alors :

$$\forall n \geq 2,\ U(n) = U(n-1) + U(n-2)$$

On recherche ensuite la solution de cette récurrence (homogène) par méthode de l'équation caractéristique : on pose une solution de type

$$U(n) = \alpha r^n$$

il vient alors : $\alpha r^n = \alpha r^{n-1} + \alpha r^{n-2} \Rightarrow r^n - r^{n-1} - r^{n-2} = 0 \Rightarrow r^2 - r - 1 = 0$

On en tire les solutions : $r_\pm = \dfrac{1 \pm \sqrt{5}}{2}$

soit $r_+ \simeq 1,6$ (le nombre d'or Φ) et $r_- \simeq -0.6$

Ainsi, $\lim_{n\to\infty} U(n) = \mathcal{O}(1,6^n)$ et donc :

$$C(n) = \mathcal{O}(1,6^n)$$

La complexité est donc exponentielle ce qui n'est évidemment pas tolérable en pratique lorsque l'on souhaite évaluer un terme de rang élevé de cette suite.

Exercice de cours: **(I.1) - n° 1.** *On dispose d'un ordinateur fonctionnant à une fréquence de $f = 1GHz$; si l'on suppose que l'unité de calcul arithmétique logique (ALU) du processeur effectue une opération élémentaire par cycle d'horloge, estimer la durée de calcul pour évaluer le $100^{ième}$ terme de la suite de Fibonacci.*

I.2 Une nouvelle approche : la programmation dynamique

a - Exemple élémentaire de la suite de Fibonacci - principe de la mémoïsation - approche "BOTTOM-UP"

La faiblesse de l'algorithme récursif naïf provient du fait qu'un nombre important de termes sont calculés plusieurs fois ; une première idée pour éliminer toute redondance de calcul est d'enregistrer chaque terme évalué, et d'y faire appel si nécessaire. On parle alors de **mémoïsation**.

On peut par exemple proposer le code suivant qui stocke tous les termes au fur et à mesure de leur évaluation dans un dictionnaire, et les utilise si nécessaire pour le calcul du terme suivant ; on évite ainsi les réévaluations inutiles. Le choix d'un dictionnaire permet de garantir la mémoïsation la plus performante possible (cf chapitre 1).

Listing III.2 – Fibonacci sans mémoïsation

```
import math as m
import time as t
def Fiborec(n):
    if n<=1:
        return n
    else:
        return Fiborec(n-1)+Fiborec(n-2)
t1=t.time()
fibo=Fiborec(40)
t2=t.time()
duree=t2-t1
print(fibo,'␣en␣un␣temps␣de␣{}␣s'.format(duree))
```

Listing III.3 – Fibonacci avec mémoïsation

```
import time as t
def Fibomem(n,dict={}):
    if n<=1:
        return n
    elif dict.get(n,0)==0: # si Fib(n) pas calculé
        dict[n]=Fibomem(n-1,dict)+Fibomem(n-2,dict)
    return dict[n]
t1=t.time()
fibo=Fibomem(40)
t2=t.time()
duree=t2-t1
print(fibo,'␣en␣un␣temps␣de␣{}␣s'.format(duree))
```

La comparaison avec le calcul sans mémoïsation est sans appel :

Calcul sans mémoïsation pour n=40 : 102334155 en un temps de 50.90771245956421 s	Calcul avec mémoïsation pour n=40 : 102334155 en un temps de 0.0 s

Enfin, on peut procéder à la dérécursivation (immédiate) de cet algorithme ce qui permet alors de réaliser le calcul par une démarche dite ASCENDANTE (par opposition à la récursivité qui procède de façon descendante) ou encore BOTTOM-UP (de bas en haut) :

Listing III.4 – Fibonacci avec mémoisation

```
import math as m
def fiboderec(n):
  T=[m.inf]*(n+1)
  T[0]=0
  T[1]=1
  for i in range(2,n-1):
    T[i]=T[i-1]+T[i-2]
  return T[n]
```

QUELQUES COMMENTAIRES :

- La complexité temporelle des versions récursive et itérative (dérécursivée) avec mémoïsation est évidemment linéaire $\mathcal{O}(n)$ puisque chaque terme n'est calculé qu'une seule fois.
- La complexité spatiale est également linéaire **ce qui peut poser un problème** si le terme à calculer est de rang élevé et que l'on cherche à préserver l'occupation de la mémoire dans des algorithmes traitant un nombre important de données.

En conclusion, le gain en complexité temporelle est donc considérable, au détriment cependant de l'occupation mémoire, écueil que va permettre de lever **la programmation dynamique**.

b - Conditions à remplir pour l'éligibilité à la programmation dynamique

Pour comprendre et construire la programmation dynamique à partir d'un algorithme récursif naïf, il est souvent utile d'exploiter un tableau bidimensionnel permettant d'exhiber la première redondance de calcul/d'appel d'un terme ; par exemple, dans le cas de suite de Fibonacci, l'évaluation de $Fib(5)$ donne le tableau suivant :

	0	1	2	3	4	5
0						
1	$Fib(0)$	$Fib(1)$				
2	$Fib(0)$	$Fib(1)$	$Fib(2)$			
3		$Fib(1)$	$Fib(2)$	$Fib(3)$		
4				$Fib(3)$	$Fib(4)$	
5						$Fib(5)$

IMPORTANT : un terme apparait sur fond rouge lorsqu'il est calculé pour la première fois, et sur fond orange lorsqu'il est de nouveau évalué. La solution optimale doit donc apparaître comme un chemin **ne passant que par des cellules rouges.**

A la lecture de ce tableau, on constate deux propriétés essentielles :

- le calcul optimal du n$^{\text{ième}}$ terme, c'est à dire sans recalcul d'aucun terme, **implique obligatoirement un calcul optimal pour tous les termes de rangs inférieurs** ; en d'autre terme, calculer de manière optimale $Fib(n)$ passe par le calcul optimal de $Fib(n-1)$, qui lui-même passe par le calcul optimal de $Fib(n-2)$ etc... On dit que le problème admet **une sous-structure optimale**.

Propriété I-1: SOUS-STRUCTURE OPTIMALE D'UN PROBLÈME

Un problème est dit posséder **une sous-structure optimale** si une solution optimale peut être construite à partir de solutions optimales de ses sous-problèmes.

- la recherche d'un chemin optimal par approche récursive montre qu'un même sous-problème apparaît plusieurs fois dans la résolution des problèmes de plus grande taille. Par exemple, le calcul de $Fib(2)$ est nécessaire dans l'évaluation de $Fib(3)$ mais aussi de $Fib(4)$, etc... On parle de **chevauchement de sous-problèmes**. C'est cette propriété qui justifie **l'usage de la mémoïsation**.

Propriété I-2: CHEVAUCHEMENT DE SOUS-PROBLÈMES

On parle de chevauchement de sous-problème lorsque des calculs aux étapes intermédiaires dans une procédure récursive sont évalués plusieurs fois. La programmation dynamique ou la mémoïsation permettent d'éliminer ce problème.

A RETENIR : les propriétés de sous-structure optimale et chevauchement de sous-problèmes sont requises pour construire une solution par programmation dynamique.

II Exemple plus technique : distance d'édition optimale ou distance de Levenshtein

II.1 Position du problème

La *distance d'édition optimale* ou encore *distance de Levenshtein* est un algorithme très utilisé permettant de comparer deux séquences de tout type, le plus souvent deux chaines de caractères `X` et `Y` ; cet algorithme renvoie **le nombre minimal de modifications à opérer sur les caractères de la chaine de caractères `X`** pour passer à la chaine de caractères `Y` parmi :

- la substitution ou remplacement, notée R_Y : on remplace un caractère de X par un autre
- l'insertion notée I : on ajoute un nouveau caractère à X
- la suppression notée S_Y : on supprime un caractère de X

Ce nombre minimal est appelé *distance de Levenshtein*.

Les moteurs de recherche disponible sur le web, tels `google.fr` ou `brave.fr`, calculent toujours la distance d'édition entre les mots saisis par l'utilisateur et ceux assez proches, appartenant à une base de données (même étymologie, même champ lexical etc....). En cas d'erreur de frappe manifeste, le moteur de recherche suggèrera une correction en proposant le mot correspondant à la distance de Lenvenshtein, c'est à dire la plus faible.

Par exemple, pour passer du mot **durite** au mot **carie** on recense :

X	Y	opération	coût
d	c	$\longrightarrow R_Y \longrightarrow$	1
u	a	$\longrightarrow R_Y \longrightarrow$	1
r	r	– – – – – – – – – –	0
i	i	– – – – – – – – – –	0
t		$\longrightarrow S_Y \longrightarrow$	1
e	e	– – – – – – – – – –	0
		coût total	**3**

soit une distance d'édition optimale de 3.

Exercice de cours: **(II.1) - n° 2.** *Déterminer la* distance de Levenshtein *entre les mots* `aluminium` *(X) et* `albumine` *(Y). On présentera la réponse sous forme d'un tableau comme dans l'exemple ci-dessus.*

On souhaite donc élaborer un algorithme efficace permettant de calculer la distance optimale d'édition.

II.2 Formulation du problème - éligibilité détaillée à la programmation dynamique

On peut facilement représenter toute solution, optimale ou non, par un tableau bidimensionnel dans lequel :

- les déplacements verticaux (vers le bas), correspondant à l'indice i, représentent une suppression supplémentaire.
- les déplacements horizontaux (vers la droite), correspondant à l'indice j, représentent une insertion supplémentaire.
- les déplacements selon la diagonale (descendante) représentent soit le remplacement d'une lettre si les caractères i et j sont différents (la valeur de la distance est incrémenté d'une unité), soit rien du tout (la valeur de la distance reste inchangée).

Par exemple le tableau correspondant à la distance **optimale** entre les mots **"carie"** et **"durite"** figure sur le schéma de gauche, alors que celui de droite représente un chemin **non optimal** ($7 > 3$! ! !) :

			$j\rightarrow$					
			0	1	2	3	4	5
				c	a	r	i	e
	0		0					
	1	d		**1**				
	2	u			**2**			
$i\downarrow$	3	r				**2**		
	4	i					**2**	
	5	t					**3**	
	6	e						**3**

		$j\rightarrow$					
			c	a	r	i	e
		0					
	d	**1**					
	u	**2**					
$i\downarrow$	r	**3**	**4**	**5**	**6**		
	i					**6**	
	t					**7**	
	e						**7**

FIGURE III.2 – Représentation de la solutions optimale et d'une solution non optimale

On appellera dans toute la suite $d(i,j)$ la distance entre les i premiers caractères du mot X et les j premiers caractères du mot Y ; si $d(i,j)$ est optimale alors on a $d(i,j) = Levenshtein(X[0:i], Y[0:j])$.

D'après cette définition, on remarque immédiatement que :

$$\begin{cases} d(0,0) = 0 \\ d(i,0) = i \\ d(0,j) = j \end{cases}$$

A partir de cette présentation, on déduit deux choses essentielles :

- la distance d'édition, optimale ou pas, se lit toujours en position $d(m,n)$, donc en bas à droite du tableau.
- Dans le chemin conduisant de manière optimale de X à Y, 4 cas de figure peuvent se présenter à chaque étape et définissent $d(i,j)$ en fonction de $d(i-1,j)$, $d(i,j-1)$, et enfin $d(i-1,j-1)$:

$$d(i,j) = \begin{cases} d(i-1,j)+1 \text{ pour la suppression du caractère } X[i] \\ d(i,j-1)+1 \text{ pour l'insertion du caractère } Y[j] \\ d(i-1,j-1)+1 \text{ si } X[i] \neq Y[j] \text{ pour le remplacement du caractère } X[i] \text{ en } Y[j] \\ d(i-1,j-1) \text{ si } X[i] = Y[j] \text{ avec aucune action entre } X[i] \text{ en } Y[j] \end{cases}$$

Le chemin optimal impose donc à chaque étape d'avoir la récurrence suivante :

$$d(i,j) = \begin{cases} min\big(d(i-1,j), d(i,j-1), d(i-1,j-1)\big) + 1 & \text{si } X[i] \neq Y[j] \\ min\big(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)\big) & \text{si } X[i] = Y[j] \end{cases}$$

Éligibilité à la programmation dynamique :

Pour s'assurer que le calcul de la distance de Levenshtein est éligible à la programmation dynamique, il faut vérifier que ce problème possède une sous-structure optimale, et présente un chevauchement de ses sous-problèmes.

■ Sous-structure optimale :

Exercice de cours: **(II.2) - n° 3.** *On suppose connue la suite des p opérations de type R_Y, S, et I_Y faisant passer de X de taille m à Y à Y de taille n, et que cette suite est* **optimale** *; en d'autres termes, cela signifie que $p = Levenshtein(X, Y)$.*
On notera X_k la chaine de caractères obtenue après la $k^{ième}$ opération de modification de X, donc $X = X_0 \overset{T_1}{\to} X_1 \overset{T_2}{\to} \dots X_{p-1} \overset{T_p}{\to} X_p = Y$.

Montrer que la suite des $p-1$ premières opérations transforme :
- *soit $X[0 : m-1]$ en $Y[0 : n-1]$*
- *soit $X[0 : m]$ en $Y[0 : n-1]$*
- *soit $X[0 : m-1]$ en $Y[0 : n]$*

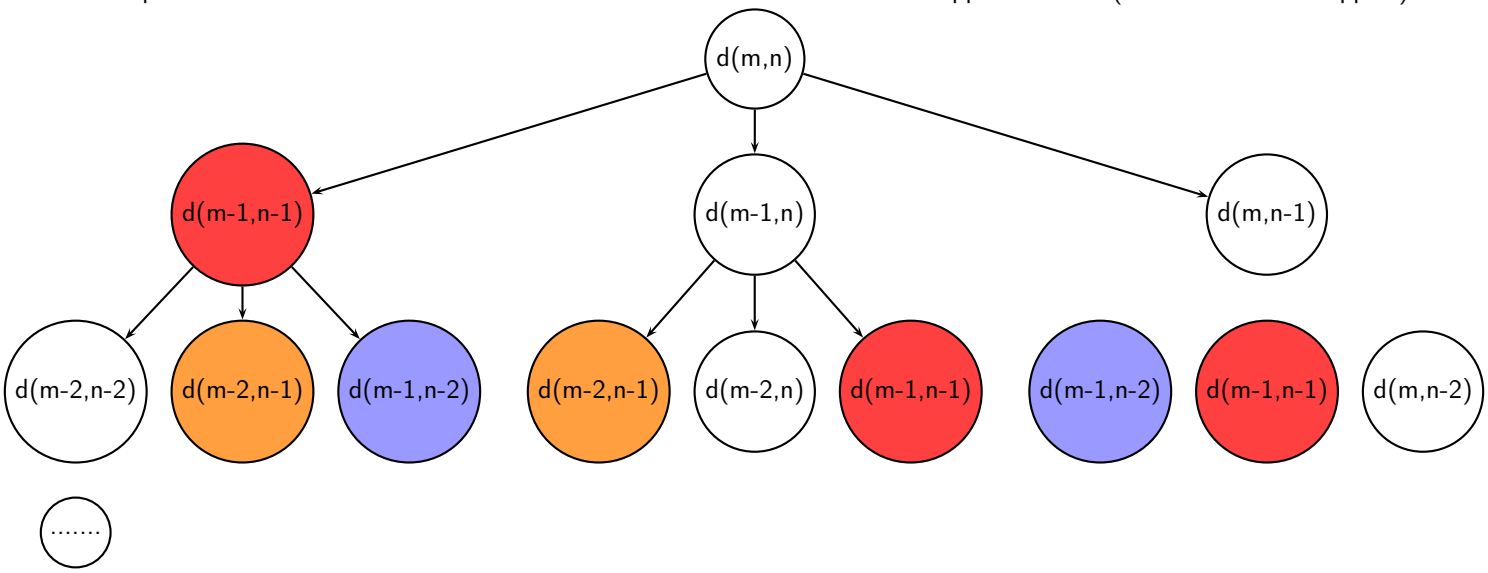
et que c'est une suite optimale (i.e. de taille minimale).

Enfin, quelque soit la dernière étape, le sous-problème à l'étape précédente est résolu de manière optimale, c'est à dire que la suite des étapes précédentes est minimale.

Conclusion : le problème présente bien une sous-structure optimale.

■ Chevauchement de sous-problèmes :

Si l'on envisage **une procédure récursive**, c'est à dire partant de $d(m, n)$, la dernière étape pour l'atteindre peut être **une suppression**, **une insertion**, ou **un remplacement**, et toutes trois devront être examinées ; il en sera de même à chaque étape antérieure. Ce constat amène à fuir l'approche récursive classique (i.e. sans mémoïsation) qui conduit donc **au lancement de 3 récursions à chaque étape** impliquant donc une complexité exponentielle.
On peut s'en convaincre très facilement en dessinant l'ébauche de l'arbre des appels récursifs (le constat est sans appel !) :



Après seulement deux niveaux de récursion explorés, on constate par exemple que les termes $d(m-1,n-1)$ et $d(m-2,n-1)$ sont chacun calculés 3 fois ! ! ! Cette procédure va conduire à une véritable explosion des appels récursifs, ce qui interdit de s'y prendre ainsi !

CONCLUSION : il y a bien chevauchement de sous-problèmes.

II.3 Codes

Comme pour l'exemple de la suite de Fibonacci, les résultats seront stockés dans un dictionnaire, structure de données la plus performante, dont les clés seront les tuples (i,j) correspondant aux indices de parcours, respectivement des chaines X et Y ; on introduit également la fonction élémentaire `minim3(x,y,z)` qui renvoie la valeur minimale de x, y, ou z.

Listing III.5 – Fonction minim3(x,y,z)

```
def minim3(x,y,z):
  mini=x
  if (y<=x and y<=z):
    mini=y
  elif (z<=x and z<=y):
    mini=z
  return mini
```

Listing III.6 – Fonction Levenshtein_dyn(X,Y)

```
def Levenshtein_dyn(X,Y):
  m,n=len(X),len(Y)
  d = { (0,0): 0 } #initialise à 0 les indices i=0 et j=0 avec un dictionnaire
  for i in range(1,m+1): #itère sur l'indice i de X
    d[i,0] = d[i-1,0] + 1 #initialise la première colonne avec i
  for j in range(1,n+1): #itère sur l'indice j de Y
    d[0,j] = d[0,j-1] + 1 #initialise la première ligne avec j
  for i in range(1,m+1):
    for j in range(1,n+1):
      if X[i-1] == Y[j-1]: # si X[i-1]=Y[j-1]
        d[i,j] = minim3(d[i-1,j]+1,d[i,j-1]+1,d[i-1,j-1])
      else:
        d[i,j] = minim3(d[i-1,j]+1,d[i,j-1]+1,d[i-1,j-1]+1)
  return d[m,n]

t1=t.time()
res=Levenshtein_dyn("approximatif","aproksimatifs")
t2=t.time()
print("Résultat:",res, "pour une durée de {} s".format(t2-t1))
```

Résultat : 4 pour une durée de 0.0 s

II.4 Reconstruction de la solution optimale

L'algorithme précédent renvoie seulement la distance de Lenvenshtein entre les deux chaines X et Y ; il peut également être intéressant de reconstruire le chemin de construction de la solution renvoyée, c'est à dire la suite des déplacements dans le tableau de calcul.

Pour cela, il suffit de modifier les deux codes précédents :

- `minim3_chem(x,y,z)` renverra non seulement la valeur minimale du triplet (x,y,z), mais également la nature du déplacement effectué depuis la précédente valeur avec comme nomenclature "B" (Bottom) pour *"vers le bas"*, "D" (Diagonal) pour *en diagonale*, et "R" (right) pour *vers la droite*.
- `Levenshtein_dyn_chem(X,Y)` renverra la distance de Levenshtein et également un tableau `chemin` de dimension $(m+1,n+1)$ tel que `chemin(i,j)` contienne le déplacement antérieur ayant permis d'atteindre cette cellule (i,j)

Exercice de cours: **(II.4) - n° 4.** *Rédiger les fonctions* `minim3_chem(x,y,z)` *et* `Levenshtein_dyn_chem(X,Y)`

Enfin, à partir du tableau `tabchemin` renvoyé, il est aisé de reconstituer l'ensemble du chemin par une démarche ascendante.

Exercice de cours: **(II.4) - n° 5.** *Rédiger la fonction* `chemin(X:str,Y:str)` $\longrightarrow$ `str` *qui, en utilisant la fonction* `Levenshtein_dyn_chem(X,Y)`*, renvoie une chaine de caractère (lettres B, R, et D) indiquant le chemin emprunté dans le tableau d'édition pour trouver la distance optimale*