PROGRAMMATION DYNAMIQUE

Introduction générale :

La programmation dynamique est une technique de programmation visant à donner les **solutions optimales** à un problème P. La méthode s'applique lorsque la résolution de P peut se faire en résolvant r sous-problèmes P_1, ..., P_r.

- On commence par chercher les solutions optimales des sous-problèmes.
- On combine ensuite ces solutions optimales pour trouver les solutions optimales de P.

Un programme dynamique peut être rédigé aussi bien en version itérative qu'en version récursive. Dans ce dernier cas, une technique particulière est utilisée par accélérer les calculs : **la mémoïsation**.

Table des matières

I. La technique de mémoïsation **2**
- **1)** Le problème de la suite de Fibonacci . . . 2
  - **a)** Méthode itérative . . . 2
  - **b)** Méthode récursive . . . 3
- **2)** La formule du binôme de Newton . . . 4
  - **a)** Méthode itérative . . . 5
  - **b)** Méthode récursive . . . 7

II. La programmation dynamique **10**
- **1)** Le problème du rendu de monnaie . . . 10
  - **a)** Formalisation mathématique du problème . . . 10
  - **b)** Unicité de la solution optimale . . . 11
  - **c)** Résolution par algorithme glouton . . . 11
  - **d)** Résolution par programmation dynamique . . . 12
  - **e)** Construction de la solution optimale . . . 16
- **2)** Distance de Levenshtein . . . 18
  - **a)** Le problème . . . 18
  - **b)** Distance de Levenshtein . . . 20
  - **c)** Algorithme force brute . . . 21
  - **d)** Recours à la programmation dynamique . . . 22
- **3)** Résumé . . . 27

III.Quelques propriétés de suites récurrentes **28**

I. La technique de mémoïsation

Prenons deux exemples qui ne concernent pas la programmation dynamique puisque *qu'il ne va pas être question de trouver une solution optimale* mais qui permettent de présenter la technique de la mémoïsation.

1) Le problème de la suite de Fibonacci

On souhaite calculer les termes de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ en utilisant la définition suivante :

$$F_0 = 0 \ ; \ F_1 = 1 \quad \text{et} \quad \forall\, n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$$

On peut dire que le problème $P =$ "calculer F_n" peut être réalisé en cherchant la solution des deux sous-problèmes $P_1 =$ "calculer F_{n-1}" et $P_2 =$ "calculer F_{n-2}".

a) Méthode itérative

Algo1 :

Si $n = 0$ ou $n = 1$ renvoyer n. Sinon créer un tableau T d'entiers de taille $n+1$ servant à stocker les F_i de 0 à n. Y placer F_0, F_1 et initialiser tous les autres éléments à 0. Pour i variant de 2 à n, faire $T[i] = T[i-1] + T[i-2]$. Renvoyer $T[n]$.

En langage Python, un tableau peut être implémenté avec une **liste** ou bien avec un **tableau numpy unidimensionnel**.

```
def Fibo1(n) :
    if n == 0 or n == 1 :
        return n
    else :
        T = (n+1)*[0]
        T[1] = 1
        for i in range(2,n+1) :
            T[i] = T[i-1] + T[i-2]
        return T[n]
```

Complexité temporelle :

Complexité spatiale :

Algo2 : version avec un dictionnaire

L'utilisation d'un dictionnaire apporte un peu de souplesse puisqu'on n'a pas besoin de créer au début un tableau de taille $n+1$. Le dictionnaire joue ce rôle et grossit au fur et à mesure que progresse l'algorithme.

```
def Fibo2(n) :
    if n == 0 or n == 1 :
        return n
    else :
        dico = {0:0 , 1:1}
        for i in range(2,n+1) :
            dico[i] = dico[i-1] + dico[i-2]
        return dico[n]
```

b) Méthode récursive

Algo3(n) : version récursive naïve

Si $n = 0$ ou bien $n = 1$, renvoyer n (condition de terminaisons des appels récursifs). Sinon calculer val = Algo3($n-1$) + Algo3($n-2$) et renvoyer val.

```
def Fibo3(n) :
    if n == 0 or n == 1 :
        return n
    else :
        val = Fibo3(n-1) + Fibo3(n-2)
    return val
```

Malheureusement, avec cette version le temps d'attente de la solution devient très long, même avec des valeurs de n assez petites : plusieurs secondes à partir de $n = 30$, de l'ordre de la minute pour $n = 40$, etc ... Le problème avec cette version est que :

- Le nombre d'appels récursifs devient "exponentiel" lorsque n est grand. Si Nb(n) est le nombre d'appels d'une des fonctions Fibo3(i) nécessaires pour trouver Fibo3(n), on peut montrer que :

$$\text{Nb}(n) \underset{\infty}{\sim} \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$$

  Avec un ordinateur cadencé à 2 GHz, chaque instruction prend au moins un temps $\tau = 5.10^{-10}$ s. Comme l'exécution de Fibo3(i) prend forcément un temps strictement supérieur à τ (et même largement supérieur !), la durée d'exécution de Fibo3(100) est de l'ordre de :

$$\Delta t > \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^{101} \times 5.10^{-10} = 1{,}15.10^{21} \text{ s} = 36\,324 \text{ milliards d'années}$$

- Un des problèmes de cette version est que l'ordinateur **refait plusieurs fois le même calcul**. Cela se voit bien en dessinant l'arbre des appels récursifs. La Figure 1 en donne l'exemple avec Fibo3(5).

On parle de **chevauchement** des sous-problèmes. Une bonne idée pour réduire le nombre d'appels récursifs est de *stocker les résultats intermédiaires dans un tableau ou bien un dictionnaire*, afin de ne pas avoir à les recalculer : c'est la **mémoïsation**[1].

1. Ce n'est pas une faute d'orthographe. Le mot est bien *mémoïsation* et non *mémorisation*.

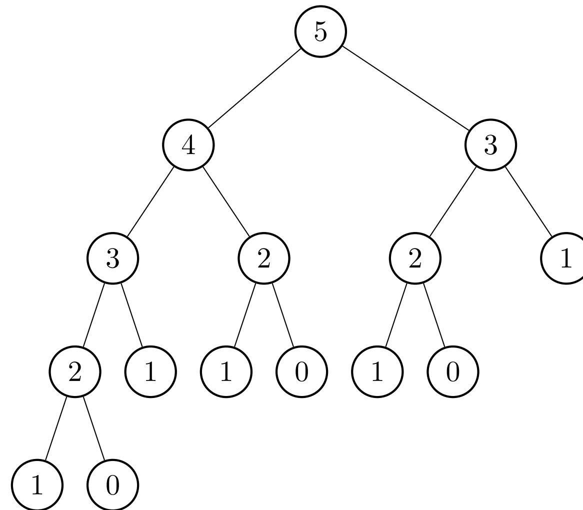

FIGURE 1 – Arbre des appels récursifs pour Fibo3(5). Fibo3(3) est appelé deux fois et Fibo3(2) trois fois

Algo4(n) : version récursive avec mémoïsation (grâce à un dictionnaire)

Si Algo4(n) déjà présent dans le dictionnaire, renvoyer Algo4(n). Sinon si $n = 0$ écrire 0 dans le dictionnaire et renvoyer 0. Sinon si $n = 1$, écrire 1 dans le dictionnaire et renvoyer 1. Sinon calculer val = Algo4($n-1$) + Algo4($n-2$), l'écrire dans le dictionnaire, puis renvoyer val.

```
dico = {} # dictionnaire vide, défini comme variable globale

def Fibo4(n) :
    if n in dico.keys() :
        return dico[n]
    elif n == 0 or n == 1 :
        dico[n] = n
        return n
    else :
        val = Fibo4(n-1) + Fibo4(n-2)
        dico[n] = val
        return val
```

Exemple :

Fibo3(20) conduit à 21 930 appels récursifs tandis que Fibo4(20) n'en provoque que 39.

2) La formule du binôme de Newton

Prenons la très célèbre formule :

$$\forall (a,b) \in \mathbb{R}^2,\ \forall n \in \mathbb{N},\ (a+b)^n = \sum_{p=0}^{n} \binom{n}{p}\, a^p\, b^{n-p}$$

Les coefficients $\binom{n}{p}$ peuvent être calculés au moyen d'une récurrence ; c'est ce qu'exprime le triangle de Pascal .

1
1 1
1 2 1
1 3 3 1
etc ...

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & 0 & \dots & 0 \\ 1 & 2 & 1 & 0 & \dots & 0 \\ 1 & 3 & 3 & 1 & \dots & 0 \\ 1 & \dots & \dots & \dots & \dots & 0 \end{bmatrix}$$

Une façon plus commode pour un informaticien de représenter cela est de créer une matrice M d'ordre $(n+1) \times (p+1)$ et d'y mettre des zéros lorsque les coefficients sont sans intérêt.

Le coefficient de M situé à l'intersection de la ligne i $(0 \leqslant i \leqslant n)$ et de la colonne j $(0 \leqslant j \leqslant n)$ est :

$$\binom{i}{j} \quad \text{et on convient que } \binom{i}{j} = 0 \text{ si } j > i$$

La relation de récurrence s'écrit :

$$\boxed{\forall (i,j) \in [\![1,n]\!] \times [\![1,p]\!],\ \binom{i}{j} = \binom{i-1}{j} + \binom{i-1}{j-1}}$$

avec (initialisation) :

$$\forall i \in [\![0,n]\!],\ \binom{i}{0} = 1 \quad \text{et} \quad \forall j \in [\![1,p]\!],\ \binom{0}{j} = 0$$

À nouveau ici, on peut dire que le problème $P =$ "calculer $\binom{n}{p}$" peut être réalisé en cherchant la solution des deux sous-problèmes $P_1 =$ "calculer $\binom{n-1}{p}$" et $P_2 =$ "calculer $\binom{n-1}{p-1}$".

a) Méthode itérative

Algo1(n,p) :

Si $p > n$ renvoyer 0. Sinon si $p = 0$ renvoyer 1. Sinon créer une matrice M de taille $(n+1) \times (p+1)$ servant à stocker les $\binom{i}{j}$, remplir la première colonne de 1 et initialiser tous les autres coefficients à 0. Pour i variant de 1 à n faire, pour j variant de 1 à p, faire $M[i,j] = M[i-1,j] + M[i-1,j-1]$. Renvoyer $M[n,p]$.

En langage Python, un matrice peut être **une liste de listes** ou bien une **matrice numpy**.

```
import numpy as np

def BinomeNewt1(n,p) :
    if p > n :
        return 0
    elif p == 0 :
        return 1
    else :
        M = np.zeros( (n+1,p+1), dtype = int)
        for i in range(n+1) :
            M[i,0] = 1
        for i in range(1,n+1) :
            for j in range(1,p+1) :
                M[i,j] = M[i-1,j] + M[i-1,j-1]
        return M[n,p]
```

Quelle est la complexité temporelle de cette fonction ? Quelle est sa complexité spatiale ?

Voici une version itérative qui utilise un dictionnaire se construisant au fur et à mesure du déroulement du processus. Les clés du dictionnaire sont les couples (i, j) (tuples) et les valeurs associées sont les $\binom{i}{j}$.

Algo2(n, p) : version itérative avec dictionnaire.

```
def BinomeNewt2(n,p) :
    if p > n :
        return 0
    elif p == 0 :
        return 1
    else :
        dico = {}
        for i in range(n+1) :
            dico[(i,0)] = 1
        for j in range(1,p+1) :
            dico[(0,j)] = 0
        for i in range(1,n+1) :
            for j in range(1,p+1) :
                dico[(i,j)] = dico[(i-1,j)] + dico[(i-1,j-1)]
        return dico[(n,p)]
```

b) Méthode récursive

Algo3(n,p) : version récursive naïve

Si $p > n$ renvoyer 0. Sinon si $p = 0$ renvoyer 1. Sinon calculer val $=$ Algo3$(n-1, p)$ $+$ Algo3$(n-1, p-1)$. Renvoyer val.

```
def BinomeNewt3(n,p) :
    if p > n :
        return 0
    elif p == 0 :
        return 1
    else :
        val = BinomeNewt3(n-1,p) + BinomeNewt3(n-1,p-1)
        return val
```

On rencontre alors les mêmes types de problèmes qu'avec la version récursive naïve de la suite de fibonacci. Le problème de cette méthode est que lorsque n augmente et que p est proche de $\lfloor n/2 \rfloor$, le temps de calcul devient long et que le nombre d'appels récursifs explose.

En effet, le nombre d'appels $\mathrm{Nb}(n, p)$ de `BinomeNewt3(n,p)` vérifie l'équation :

$$\mathrm{Nb}(n, p) = 1 + \mathrm{Nb}(n-1, p) + \mathrm{Nb}(n-1, p-1)$$

et on montre facilement que :

$$\boxed{\forall n \in \mathbb{N},\ \forall p \in \mathbb{N},\ \mathrm{Nb}(n, p) \geqslant \binom{n}{p} \quad \text{en convenant que} \quad \binom{n}{p} = 0 \text{ si } p > n}$$

Il suffit par exemple de montrer par récurrence sur n que l'assertion :

$$A(n) : \forall p \in \mathbb{N},\ \mathrm{Nb}(n, p) \geqslant \binom{n}{p}$$

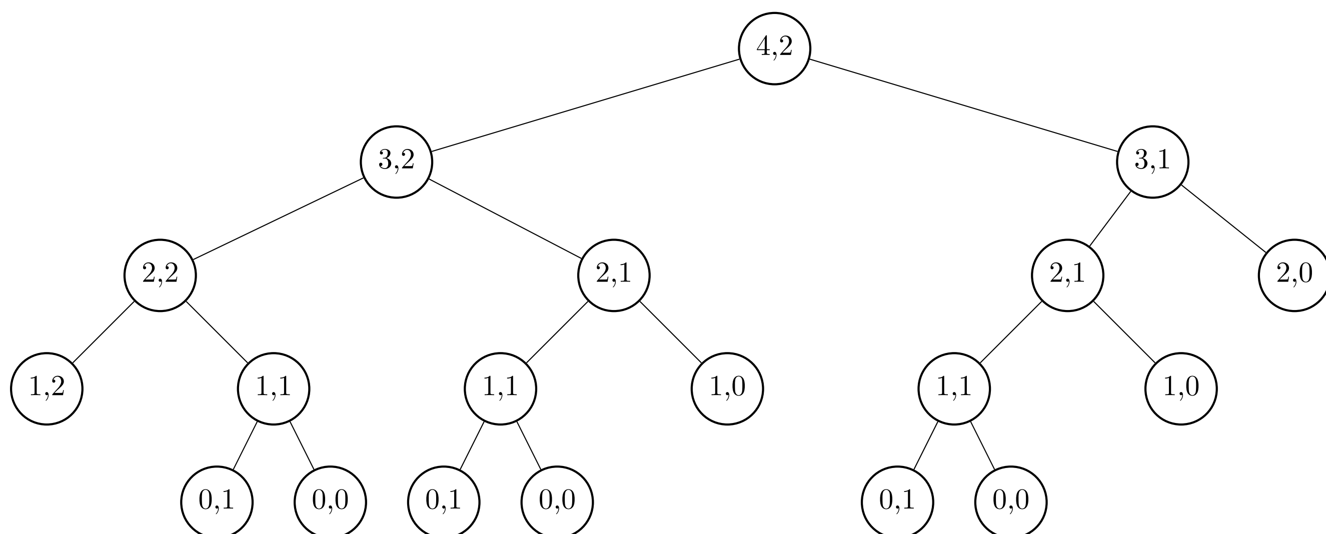

FIGURE 2 – Arbre des appels récursifs provoqués par BinomeNewt3(4,2)

est toujours vraie.

En outre, un même calcul peut être effectué plusieurs fois. On peut le voir facilement en dessinant l'arbre des appels récursifs de `BinomeNewt3(4,2)` : il y a à nouveau un **chevauchement** des sous-problèmes.

Prenons le cas critique où $n = 2p$ et utilisons la formule de Stirling pour donner un ordre de grandeur asymptotique du nombre d'appels lorsque p devient très grand. La formule est :

$$n! \underset{\infty}{\sim} \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Ici de même une technique de mémoïsation permet de réduire le nombre de calculs. On *stocke les résultats intermédiaires dans un dictionnaire* dont les clés sont les couples (i, j) (tuples) et les valeurs $\binom{i}{j}$. Cela donne :

Algo4(n,p) : version récursive avec mémoïsation (à l'aide d'un dictionnaire)

Si Algo4(n, p) déjà présent dans le dictionnaire, renvoyer Algo4(n, p). Sinon si $p > n$, écrire 0 dans le dictionnaire et renvoyer 0. Sinon si $p = 0$ écrire 1 dans le dictionnaire et renvoyer 1. Sinon calculer val = Algo4$(n-1, p)$ + Algo4$(n-1, p-1)$, l'écrire dans le dictionnaire, puis renvoyer val.

```
dico = {} # dictionnaire vide défini comme variable globale

def BinomeNewt4(n,p) :
   if (n,p) in dico.keys() :
      return dico[(n,p)]
   elif p > n :
      dico[(n,p)] = 0
      return 0
   elif p == 0 :
      dico[(n,p)] = 1
      return 1
   else :
      val = BinomeNewt4(n-1,p) + BinomeNewt4(n-1,p-1)
      dico[(n,p)] = val
      return val
```

Exemple :

Le nombre d'appels effectués par `BinomeNewt3(16,8)` est de 48 619 avant d'afficher le résultat (12 870). Avec mémoïsation, ce nombre d'appels est réduit à 145.

II. La programmation dynamique

Venons-en maintenant à notre objet d'étude qui est la programmation dynamique. Plutôt que de tenir des propos généraux indigestes, nous allons prendre quelques exemples. On a déjà dit que cette méthode de programmation s'intéresse à des problèmes d'optimisation lorsqu'un problème P peut être résolu en commençant par résoudre r sous-problèmes P_1, ..., P_r.

1) Le problème du rendu de monnaie

On dispose de pièces de monnaie et de billets, par exemple dans le système monétaire européen nous avons des pièces de 1 et 2 euros, puis des billets de 5, 10, 20, 50, 100, 200 et 500 euros. Dans la suite, nous appellerons indifféremment "pièces" soit de vraies pièces de monnaie, soit des billets.

On souhaite rendre une somme d'argent s avec ces pièces mais en utilisant le moins de pièces possible, sachant que l'on n'est pas limité et qu'on dispose de toutes les pièces nécessaires.

a) Formalisation mathématique du problème

Il y a 9 types de pièces que nous numérotons de 1 à 9. Appelons m_i le montant des pièces de type n°i. Nous avons donc :

$$m_1 = 1,\ \ m_2 = 2,\ \ m_3 = 5,\ \ m_4 = 10,\ \ m_5 = 20,\ \ m_6 = 50,\ \ m_7 = 100,\ \ m_8 = 200 \quad \text{et} \quad m_9 = 500$$

Si on appelle n_i le nombre de pièces de montant m_i dans la somme s à rendre et n le nombre de pièces rendues, nous aurons :

$$s = \sum_{i=1}^{9} n_i\, m_i \quad (1) \quad \text{et} \quad n = \sum_{i=1}^{9} n_i \quad (2)$$

Fixons-nous une somme s à rendre et appelons $E(s)$ l'ensemble des solutions de (1) :

$$E(s) = \left\{ (n_1, n_2, \dots, n_9) \in \mathbb{N}^9 \;\middle|\; s = \sum_{i=1}^{9} n_i\, m_i \right\}$$

Il est facile de vérifier que $E(s) \neq \varnothing$ pour tout $s \in \mathbb{N}$. Soit alors $V(s)$ l'ensemble :

$$V(s) = \left\{ n = \sum_{i=1}^{9} n_i \;\middle|\; (n_1, n_2, \dots, n_9) \in E(s) \right\} \subset \mathbb{N}$$

En tant que partie non vide de $\mathbb{N}$, $V(s)$ admet un plus petit élément que nous notons $n_{\min}(s)$. De ce fait, nous pouvons introduire l'ensemble :

$$E^*(s) = \left\{ (n_1, n_2, \dots, n_9) \in E(s) \;\middle|\; \sum_{i=1}^{9} n_i = n_{\min}(s) \right\}$$

ou encore :

$$E^*(s) = \left\{ (n_1, n_2, \dots, n_9) \in \mathbb{N}^9 \;\middle|\; s = \sum_{i=1}^{9} n_i\, m_i \quad \text{et} \quad \sum_{i=1}^{9} n_i = n_{\min}(s) \right\}$$

$E^*(s)$ est appelé ensemble des **solutions optimales** de l'équation (1). Il s'agit d'un sous-ensemble de l'ensemble $E(s)$ des solutions de (1).

b) Unicité de la solution optimale

On peut montrer que, dans le système des euros, pour chaque somme s, il n'y a qu'une seule solution optimale ce qui signifie que $E^*(s)$ est un singleton ou encore que $\mathrm{Card}(E^*(s)) = 1$.

Exercice

1. Quelles sont les solutions optimales pour $s = 1$, $s = 2$, $s = 3$, $s = 4$?
2. Supposons qu'il n'y ait qu'une seule solution optimale pour une somme $s > 0$ donnée. Dans quelles limites peuvent varier les nombres de pièces de 1 euros, 2 euros, 5 euros, etc ..., 500 euros qui définissent cette solution ?
3. On considère la somme $s+1$ obtenue en ajoutant 1 euro à s. Expliciter comment construire une solution optimale et montrer qu'il n'y a qu'une seule possibilité. Conclure.

* * * * * * * * * * * * * **

Nous nous proposons maintenant d'étudier des algorithmes qui répondent aux deux questions suivantes :

1) Pour une somme s donnée, que vaut $n_{\min}(s)$?
2) Connaissant $n_{\min}(s)$, trouver la solution optimale, c'est à dire l'unique éléments de $E^*(s)$.

Il y a deux types d'algorithmes destinés à résoudre ce genre de problème :

- Les algorithmes gloutons vus en MPSI
- La programmation dynamique

c) Résolution par algorithme glouton

Un algorithme glouton résout le problème étape par étape :

- à chaque étape, il donne une solution optimale (ce qu'on peut appeler un optimum local)
- Il évalue ce qui lui reste à résoudre et passe à l'étape suivante.
- Les solutions données aux étapes précédentes ne sont jamais remises en question.

Algo Glouton :

Prendre la pièce de plus grande valeur m inférieure à s. Rendre le nombre maximal n de pièces de valeur m tel que $nm \leqslant s$. Calculer ce qui reste à rendre : $s - nm$. Recommencer l'étape précédente jusqu'à ce que toute la somme s ait été rendue.

On en donne une version avec dictionnaire pour stocker la valeur et le nombre de pièces rendues. Le dictionnaire est formé des couples $(m : n)$ (les clés sont donc les montants m de chaque pièce).

```
Monnaie = [500, 200, 100, 50, 20, 10, 5, 2, 1]
Rendu = {}

for m in Monnaie :
   if m <= s :
      n = s//m
      Rendu[m] = n
      s = s % m

n_min = 0
   for m in Rendu.keys() :
      n_min += Rendu[m]
```

Cet algorithme glouton trouve-t-il toujours une solution optimale ? (qu'il trouve une solution de devrait pas poser de question, mais est-elle optimale ?). *Cela dépend du jeu de pièces de monnaie qu'on possède.*

Par exemple imaginons un pays ne possédant que les pièces $[7, 5, 1]$. Que renvoie l'algorithme glouton si $s = 11$? Quelle est la solution optimale ?

d) Résolution par programmation dynamique

On s'intéresse pour commencer au problème suivant : pour une somme s donnée, quel est le nombre minimal $n_{\min}(s)$ de pièces à rendre, c'est à dire le plus petit élément de $V(s)$.

On ne s'intéresse pas pour le moment à trouver la solution optimale qui réalise ce minimum (on la construira plus tard).

D'après la section **1)** on sait que $n_{\min}(s)$ est le plus petit élément de $V(s)$ défini par :

$$V(s) = \left\{ n = \sum_{i=1}^{9} n_i \;\middle|\; (n_1, n_2, \ldots, n_9) \in \mathbb{N} \quad \text{et} \quad s = \sum_{i=1}^{9} n_i\, m_i \right\}$$

Exercice :

1. De quel(s) élément(s) est composé $V(0)$? Quel est alors son plus petit élément ?
2. Supposons maintenant que $s > 0$. Montrer qu'on a :

$$V(s) = \bigcup_{\substack{1 \leqslant k \leqslant 9 \\ m_k \leqslant s}} \{\, 1 + n \mid n \in V(s - m_k) \,\}$$

On peut faire un raisonnement purement mathématique (en montrant la double inclusion) ou bien faire un raisonnement plus intuitif en remarquant que pour rendre la somme s, il faut déja commencer par rendre **une pièce** : si on choisit une pièce d'un montant $m_k \leqslant s$, il ne restera alors plus qu'à rendre la somme $s - m_k < s$.

3. En déduire que :

$$\forall\, s > 0,\; n_{\min}(s) = \min_{\substack{1 \leqslant k \leqslant 9 \\ m_k \leqslant s}} (1 + n_{\min}(s - m_k)) \quad \text{et} \quad n_{\min}(0) = 0 \qquad (*)$$

On pourra utiliser la propriété suivante (qu'on pourra chercher à démontrer) :

Si A et B sont deux parties non vides de $\mathbb{N}$ et que $a = \min A$ et $b = \min B$ alors :

$$\min\,(A \cup B) = \min(a, b)$$

ce qui peut se généraliser immédiatement par récurrence à toute réunion **finie** de parties de $\mathbb{N}$

L'équation $(*)$ peut être vue comme une sorte de récurrence sur $n_{\min}(s)$: l'expression de $n_{\min}(s)$ est donnée en fonction de celles de $n_{\min}(s - m_k)$; on peut dire que le problème $P =$ "trouver $n_{\min}(s)$" dépend de la résolution de k sous-problèmes $P_k =$ "trouver $n_{\min}(s-m_k)$"

De plus, chacun de ces sous-problèmes est *optimal* : on parle de **propriété de sous-structure optimale**.

Un programme résolvant l'équation $(*)$ peut être écrit aussi bien en version récursive qu'en version itérative.

Algo1(s) : version récursive naïve

Si $s = 0$ renvoyer 0. Sinon pour chaque $m_k \leqslant s$, calculer 1 + Algo1$(s - m_k)$ et stocker les résultats dans un tableau L ; renvoyer le plus petit élément de L.

Dans le langage Python, le tableau L sera implémenté par une **liste**.

```
Monnaie = [1,2,5,10,20,50,100,200,500]

def n_min1(s) :
   if s == 0 :
      return 0
   else :
      L = []
      for m in Monnaie :
         if m <= s :
            N = 1 + n_min1( s - m )
            L.append(N)
         else :
            break
      val = min(L)
      return val
```

Exercice 1. Réécrire cette fonction avec une boucle `while` à la place de la boucle `for` et sans utiliser de `break`.

Exercice 2. Écrire une version de cette fonction qui trouve le minimum "à la volée", c'est à dire sans utiliser de liste `L` ni la fonction `min` du langage Python. On peut utiliser la technique suivante :

Techniques de programmation en Python : pour moi = occasion de parler des fonctions `min`, `max`, des instructions `break`, `continue`, `pass` et d'introduire `math.inf` ou `float("inf")`

Malheureusement, cette technique récursive naïve prend assez rapidement beaucoup de temps. Encore une fois, ceci est dû au fait que les appels récursifs deviennent assez rapidement nombreux et conduisent à refaire plusieurs fois le même calcul. On peut le voir par exemple en dressant l'arbre des appels récursifs de `n_min1(6)`.

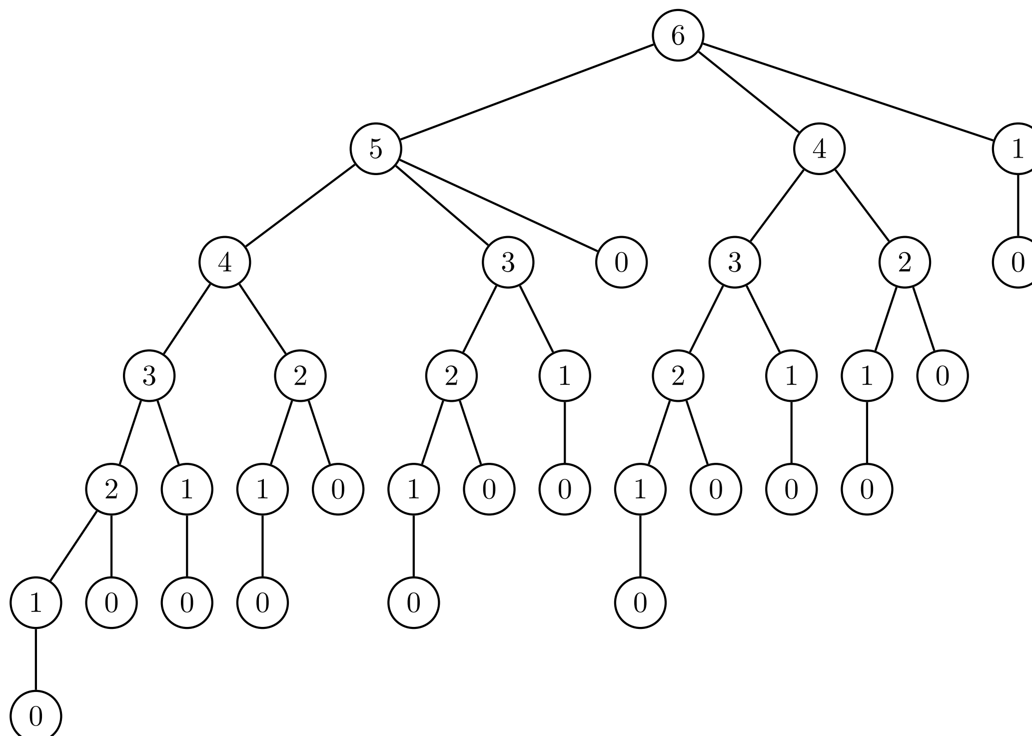

FIGURE 3 – Arbre des appels récursifs de n_min1(6)

Il y a donc un **chevauchement** des sous-problèmes et une technique de mémoïsation est nécessaire pour accélérer les calculs. On va donc utiliser un dictionnaire dont les éléments seront les couples (s : $n_\min(s)$) pour stocker les résultats et ne pas avoir à les recalculer tout le temps.

Algo2(s) : version récursive avec mémoïsation (on utilise un dictionnaire)

Si Algo2(s) est déjà présent dans le dictionnaire, renvoyer Algo2(s). Sinon si $s = 0$, écrire 0 dans le dictionnaire et renvoyer 0. Sinon pour chaque $m_k \leqslant s$:

— si Algo2($s - m_k$) est déjà présent dans le dictionnaire, stocker sa 1 + sa valeur dans un tableau L,

— sinon appeler Algo2($s - m_k$) et stocker 1 + valeur renvoyée dans le tableau L ;

trouver le plus petit élément de L et l'écrire dans le dictionnaire ; renvoyer ce plus petit élément.

```
Monnaie = [1,2,5,10,20,50,100,200,500]
dico = {}

def n_min2(s) :
   if s in dico.keys() :
      return dico[s]
   elif s == 0 :
      dico[s] = 0
      return 0
   else :
      L = []
      for m in Monnaie :
         if m <= s :
            if s-m in dico.keys() :
               N = 1 + dico[s-m]
            else :
               N = 1 + n_min2( s - m )
            L.append(N)
         else :
            break
      val = min(L)
      dico[s] = val
      return val
```

Algo3(s) : version itérative avec tableau T de taille $s+1$: i est l'indice des alvéoles de T et $T[i]$ contient $n_{\min}(i)$.

Si $s=0$ renvoyer 0. Sinon créer un tableau T de taille $s+1$ et le remplir avec des zéros. Pour i variant de 1 à s faire : trouver le plus petit des $1+T[i-m]$ lorsque m varie dans Monnaie de sorte que $m \leqslant i$ et mettre le résultat dans $T[i]$. Renvoyer $T[s]$.

```
Monnaie = [1,2,5,10,20,50,100,200,500]

def n_min3(s) :
   if s == 0 :
      return 0
   else :
      T = (s+1)*[0]
      for i in range(1,s+1) :
         L = []
         for m in Monnaie :
            if m <= i :
               N = 1 + T[i-m]
               L.append(N)
            else :
               break
         T[i] = min(L)
      return T[s]
```

Algo4(s) : version itérative avec un dictionnaire dont les éléments sont les couples (s : n_min(s)) (les clés du dictionnaire sont donc les sommes s)

Si $s = 0$ renvoyer 0. Sinon créer un dictionnaire `dicoMin` et l'initialiser à `{0:0}`. Pour i variant de 1 à s : trouver le plus petit des 1 + `dicoMin`$[i-m]$ lorsque m varie dans Monnaie de sorte que $m \leqslant i$ et mettre le résultat dans `dicoMin`$[i]$. Renvoyer `dicoMin`$[s]$.

```
Monnaie = [1,2,5,10,20,50,100,200,500]

def n_min4(s) :
   if s == 0 :
      return 0
   else :
      dicoMin = {0:0}
      for i in range(1,s+1) :
         L = []
         for m in Monnaie :
            if m <= i
               N = 1 + dicoMin[i - m]
               L.append(N)
            else :
               break
         dicoMin[i] = min(L)
      return dicoMin[s]
```

e) Construction de la solution optimale

Pour le moment nous n'avons que $n_{\min}(s)$ pour tout $s \in \mathbb{N}$. On souhaite maintenant connaître la solution optimale, c'est à dire l'unique élément de $E^*(s)$ pour chaque somme s. Autrement dit :

Pour toute somme s à rendre, on souhaite obtenir la liste des pièces qui la constitue.

Dans ce but, il est nécessaire de compléter les algorithmes précédents en y ajoutant une *information supplémentaire* qui va nous permettre de **construire** cette solution optimale.

L'idée est que chacun des 4 algorithmes précédents renvoie non seulement n_min(s) mais aussi le montant particulier m_p qui a rendu minimal $1 + \underset{\substack{1 \leqslant k \leqslant 9 \\ m_k \leqslant s}}{n_{\min}}(s - m_k)$.

À titre d'exemples, voici les versions modifiées de Algo1 et Algo4. Elles utilisent la méthode `T.index` de la liste `T` (pour moi : c'est ici l'occasion de parler de cette méthode...)

Version modifiée de l'algorithme récursif naïf :

```
Monnaie = [1,2,5,10,20,50,100,200,500]

def n_min1_modif(s) :
   if s == 0 :
       return 0,0
   else :
      T = []
      for m in Monnaie :
         if m <= s :
            N = 1 + n_min1_modif( s - m )[0]
            T.append(N)
         else :
            break
      val = min(T)
      mp = Monnaie[ T.index(val) ]
      return val, mp
```

Version modifiée de l'algorithme itératif avec dictionnaire

```
Monnaie = [1,2,5,10,20,50,100,200,500]

def n_min4_modif(s) :
   if s == 0 :
      return 0,0
   else :
      dicoMin = {0:0}
      for somme in range(1,s+1) :
         T = []
         for m in Monnaie :
            if m <= somme
               N = 1 + dicoMin[somme - m]
               T.append(N)
            else :
               break
         val = min(T)
         dicoMin[somme] = val
         mp = Monnaie[ T.index(val) ]
      return dicoMin[s], mp
```

Exercice :

Re-écrire ces fonctions sans utiliser la méthode `L.index`. Écrire les versions modifiées des Algo2 et Algo3.

La fonction `construire(s)` ci-dessous construit la solution optimale en stockant les pièces rendues dans une liste (on pourrait aussi utiliser un dictionnaire). On en donne une version récursive et une version itérative :

Version récursive de `construire` :

```
def construire_rec(s) :
  if s == 0 :
    return []
  else :
    n,m = n_min1_modif(s)
    liste_pieces = [m] + construire(s-m)
    return liste_pieces
```

Remarque :

Il vaut mieux utiliser `n_min2_modif`, `n_min3_modif` ou `n_min4_modif` pour une raison de complexité temporelle mais n'importe laquelle de ces trois fonctions convient puisqu'elles renvoient le même type de valeurs.

Version itérative de `construire` :

```
def construire_itera(s) :
  if s == 0 :
    return []
  else :
    liste_pieces = []
    s1 = s
    while s1 != 0 :
      n,m = n_min1_modif(s1)
      liste_pieces.append(m)
      s1 = s1 - m
    return liste_pieces
```

2) Distance de Levenshtein

a) Le problème

Considérons deux chaînes de caractères ch_1 et ch_2, c'est à dire deux suites finies de caractères de la forme $a_1a_2\dots a_n$ et $b_1b_2\dots b_p$ où les a_i et les b_j sont des caractères (c'est à dire des symboles typographiques) qui appartiennent à un alphabet.

On souhaite passer de ch_1 à ch_2 grâce à l'une des trois opérations suivantes :

1. **Insertion** d'un caractère de ch_2 dans ch_1
2. **Suppression** d'un caractère de ch_1
3. **Remplacement** d'un caractère de ch_1 par un caractère de ch_2

Prenons l'exemple de ch_1 = AGORRYTNES et ch_2 = ALGORITHMES

Mise en forme du problème : l'alignement

Une bonne approche du problème consiste à commencer par ce qu'on appelle un alignement des deux chaînes de caractères. On place ch_1 au dessus de ch_2 et on se permet éventuellement

de créer des espace **à gauche**, **à droite** ou **à l'intérieur** de ch_1 ou de ch_2 pour des raisons qui vont bientôt apparaître. Traditionnellement, ces espace sont *représentés par des traits*[2].

Exemples d'alignements possibles :

A	G	O	R	R	Y	T	N	E	S	–
A	L	G	O	R	I	T	H	M	E	S

A	G	O	R	R	Y	T	N	–	E	S
A	L	G	O	R	I	T	H	M	E	S

A	–	G	O	R	R	Y	T	N	E	S
A	L	G	O	R	I	T	H	M	E	S

A	–	G	O	R	R	Y	T	N	–	E	S
A	L	G	O	R	–	I	T	H	M	E	S

A	G	O	R	R	Y	T	N	–	E	S	–	–
–	–	A	L	G	O	R	I	T	H	M	E	S

Le point important est que chaque caractère de ch_1 se trouve soit face à un caractère de ch_2, soit face à un trait $-$ de ch_2.

De même, il faut que chaque caractère de ch_2 se trouve soit face à un caractère de ch_1, soit face à un trait $-$ de ch_1.

Autrement dit on ne peux jamais avoir deux traits face à face : cela voudrait dire qu'on a créé un espace en trop dans les deux chaînes et qu'on peut le supprimer sans rien changer au problème.

En comptant les traits, on se retrouve donc avec deux chaînes de même longueur et, si on se concentre sur ce qui se passe à une position donnée, on peut rencontrer 4 situations différentes :

2. On suppose donc que le trait $-$ ne fait pas partie de l'alphabet.

.

ch_1 :	$\dots\ x\ \dots$	ch_1 :	$\dots\ x\ \dots$	ch_1 :	$\dots\ x\ \dots$	ch_1 :	$\dots\ -\ \dots$
ch_2 :	$\dots\ y\ \dots$	ch_2 :	$\dots\ x\ \dots$	ch_2 :	$\dots\ -\ \dots$	ch_2 :	$\dots\ y\ \dots$

où x et y désignent deux caractères de l'alphabet.

Étant donné un alignement, pour passer de ch_1 à ch_2 selon cet alignement, on réalise les opérations suivantes :

Cas 1 : remplacer x par y.
Cas 2 : ne rien faire.
Cas 3 : supprimer x
Cas 4 : insérer y

et on recommence la même chose pour toutes les positions.

Pour chaque alignement a, on réalise donc une suite d'opérations qui transforment progressivement ch_1 en ch_2. Cependant, on attribue un coût à chaque opération :

1 pour insérer, supprimer et remplacer	0 si on ne fait rien

On note $C(a)$ le **coût total** de la transformation de ch_1 en ch_2 (c'est à dire la somme des coûts de chaque opération) pour un alignement a donné.

Exercice : calculer le coût total de la transformation de AGORRYTNES en ALGORITHMES pour chaque alignement de l'exemple précédent.

b) Distance de Levenshtein

Étant données deux chaînes de caractères ch_1 et ch_2, soit A l'ensemble des alignements permis, compte-tenu des règles énoncées ci-dessus [3].

Définition. *Distance de Levenshtein*

Étant données deux chaînes de caractères ch_1 et ch_2 on appelle **distance de Levenshtein** [a] (ou *distance d'édition*) entre ch_1 et ch_2 et on note Lev(ch_1,ch_2) le nombre entier défini par :

$$\mathrm{Lev}(\mathrm{ch}_1, \mathrm{ch}_2) = \min_{a \in A} C(a)$$

a. Cette distance fut introduite pour la première fois par Vladimir Levenshtein, informaticien russe, en 1965.

Autrement dit, Lev(ch_1,ch_2) est le ***nombre minimal d'opérations*** *à effectuer pour transformer ch_1 en ch_2.*

Utilités :

- Orthographe : réalisation d'un correcteur orthographique, reconnaissance optique de caractères.
- Linguistique : détection de la proximité ente deux langues.

3. On peut par exemple numéroter chaque alignement en commençant par 1 et en allant jusqu'au nombre maximum n d'alignements possibles pour ces deux chaînes. On aura dans ce cas $A = \{1, 2, ..., n\}$.

- Bio-informatique : similarité de séquences d'ADN

Conventions et notations :

- Le nombre de caractères d'une chaîne ch (sans les traits d'alignement !) sera noté |ch|. Par exemple si ch $= a_1 a_2 \dots a_n$ alors $|\text{ch}| = n$.
- La chaîne vide (c'est à dire la chaîne ne contenant aucun caractère, représentée par `""` en langage Python) sera notée ε. Par définition : $|\varepsilon| = 0$.

Exercice 1 : vérifier que, pour toute chaîne ch, $\text{Lev}(\varepsilon, \text{ch}) = \text{Lev}(\text{ch}, \varepsilon) = |\text{ch}|$.

Exercice 2 :

Soit E l'ensemble des chaînes de caractères formées sur un alphabet donné. Montrer que Lev est bien une *distance* sur E.

c) Algorithme force brute

Pour déterminer la distance de Levenshtein entre deux chaînes de caractères ch_1 et ch_2, un algorithme force brute :

1. détermine tous les alignements possibles entre les deux chaînes ;
2. calcule le coût total de chaque alignement ;
3. en déduit le coût total minimum $= \text{Lev}(\text{ch}_1,\text{ch}_2)$.

La complexité d'un tel algorithme le rend totalement inefficace, voire impossible à mettre en œuvre. Soient en effet deux chaînes $\text{ch}_1 = a_1 a_2 \dots a_i$ et $\text{ch}_2 = b_1 b_2 \dots b_j$. Combien existe-t-il d'alignement permis ?

Soit $f(i,j)$ ce nombre : il est possible de le calculer par récurrence. Supposons $j \geqslant 1$ et $j \geqslant 1$. On a :

$f(i,j) =$ [Nombre d'alignements de $a_1 ... a_{i-1}$ et de $b_1 ... b_{j-1}$] $\begin{matrix} a_i \\ b_j \end{matrix}$

$+$ [Nombre d'alignements de $a_1 ... a_i$ et de $b_1 ... b_{j-1}$] $\begin{matrix} - \\ b_j \end{matrix}$

$+$ [Nombre d'alignements de $a_1 ... a_{i-1}$ et de $b_1 ... b_j$] $\begin{matrix} a_i \\ - \end{matrix}$

et donc :

$$\forall (i,j) \in \mathbb{N}^* \times \mathbb{N}^*,\ f(i,j) = f(i-1,j-1) + f(i,j-1) + f(i-1,j)$$

avec :

$$\forall i \in \mathbb{N},\ f(i,0) = 1 \quad \text{et} \quad \forall j \in \mathbb{N},\ f(0,j) = 1$$

On crée une matrice F d'ordre $(n+1) \times (p+1)$ dont le coefficient F_{ij} situé à l'intersection de la ligne i et de la colonne j est $f(i,j)$ et on la remplit grâce à la récurrence précédente. Voici

le résultat pour $n = p = 5$:

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 5 & 7 & 9 & 11 \\ 1 & 5 & 13 & 25 & 41 & 61 \\ 1 & 7 & 25 & 63 & 129 & 231 \\ 1 & 9 & 41 & 129 & 321 & 681 \\ 1 & 11 & 61 & 231 & 681 & 1683 \end{bmatrix}$$

On peut montrer que :

$$f(n,n) \underset{\infty}{\sim} \sqrt{n}\,\left(1+\sqrt{2}\right)^{2n+1}$$

Exemple : $f(1000, 1000) \approx 2{,}7 \times 10^{767} \gg$ nombre d'atomes dans l'univers estimé à 10^{80}. Il n'y aurait donc même pas assez de ressources matérielles dans tout l'univers pour fabriquer les cellules mémoires capables de stocker ces alignements...

d) Recours à la programmation dynamique

α) Formalisation mathématique

Définition. *Préfixe d'une chaîne de caractères*

Soit ch une chaîne de caractères. On appelle **préfixe** de ch toute chaîne de caractère ch_p vérifiant :

$$\exists\,\mathrm{ch}',\ \mathrm{ch} = \mathrm{ch}_p + \mathrm{ch}'$$

où + est l'opération de concaténation des chaînes de caractères. On appelle **longueur** du préfixe ch_p le nombre de caractères de ch_p, c'est à dire $|\mathrm{ch}_p|$.

Exemple : "Bon" est un préfixe (de taille 3) de ch = "Bonjour" puisque "Bonjour" = "Bon" + "jour".

Exercice 3 :

1. Montrer que toute chaîne ch est un préfixe d'elle-même (il s'agit du préfixe de taille maximale).
2. La chaîne vide ε est un préfixe (de taille nulle) de n'importe quelle chaîne de caractères.

De façon générale, la longueur des préfixes d'une chaîne ch est comprise entre 0 et $|\mathrm{ch}|$. Pour un entier i donné avec $0 \leqslant i \leqslant |\mathrm{ch}|$, il existe **un et un seul préfixe** de ch ayant i comme longueur et nous le notons $\mathrm{pref}_{\mathrm{ch}}(i)$

* * * * * * * * * * * * * *

La possibilité de recourir à la programmation dynamique vient de la propriété suivante.

Soient $\mathrm{ch}_1 = a_1 \dots a_n$ et $\mathrm{ch}_2 = b_1 \dots b_p$ deux chaînes de caractères de longueurs respectives $n > 0$ et $p > 0$. Soient $\mathrm{pref}_{\mathrm{ch}_1}(i)$ et $\mathrm{pref}_{\mathrm{ch}_2}(j)$ les préfixes de ch_1 et de ch_2 de longueurs respectives i et j, avec $0 \leqslant i \leqslant n$ et et $0 \leqslant j \leqslant p$. On pose :

$$\boxed{d(i,j) = \mathrm{Lev}(\,\mathrm{pref}_{\mathrm{ch}_1}(i), \mathrm{pref}_{\mathrm{ch}_2}(j)\,)}$$

Alors $d(i, j)$ possède les propriétés suivantes :

1. $\forall\, i \in [\![0, n]\!],\ d(i, 0) = i$
2. $\forall\, j \in [\![0, p]\!],\ d(0, j) = j$

3. $\forall (i,j) \in [\![1,n]\!] \times [\![1,p]\!],\ d(i,j) = \min \begin{cases} 1 + d(i-1,j) \\ 1 + d(i,j-1) \\ \left| \begin{array}{ll} 1 + d(i-1,j-1) & \text{si} \quad a_i \neq b_j \\ d(i-1,j-1) & \text{si} \quad a_i = b_j \end{array} \right. \end{cases}$

   où a_i et b_j sont respectivement les derniers caractères de $\text{pref}_{\text{ch}_1}(i)$ et de $\text{pref}_{\text{ch}_2}(j)$.

Preuve : sur feuille à part.

À nouveau on voit apparaître une "sorte de récurrence" dans laquelle la solution optimale "trouver $d(i,j)$" fait appel aux solutions optimales des 3 sous-problèmes "trouver $d(i-1,j)$", "trouver $d(i,j-1)$" et "trouver $d(i-1,j-1)$" ; il y a bien une propriété de **sous-structure optimale** qui se dégage.

β) Versions récursives de l'algorithme

On utilise la relation de récurrence précédente en partant de la fin, c'est à dire de $d(n,p)$ (cela revient à faire $i = n$ et $j = p$ dans ces relations.)

Algo1(ch_1, ch_2) : version récursive naïve

- Calculer $n = |\text{ch}_1|$ et $p = |\text{ch}_2|$
- Si $n = 0$ renvoyer p
- Sinon si $p = 0$ renvoyer n
- Sinon trouver le minimum parmi :
  - 1 + Algo1($\text{pref}_{\text{ch}_1}(n-1)$,$\text{ch}_2$),
  - 1 + Algo1(ch_1,$\text{pref}_{\text{ch}_2}(p-1)$)
  - (1 + Algo1($\text{pref}_{\text{ch}_1}(n-1)$,($\text{pref}_{\text{ch}_2}(p-1)$) si $a_n \neq b_p$ ou Algo1($\text{pref}_{\text{ch}_1}(n-1)$,($\text{pref}_{\text{ch}_2}(p-1)$) si $a_n = b_p$) ;

  Renvoyer ce minimum.

```
def lev1(ch1,ch2) :
   n,p = len(ch1), len(ch2)
   if n == 0 :
      return p
   elif p == 0 :
      return n
   else :
      val1 = 1 + lev1(ch1[0:n-1],ch2)
      val2 = 1 + lev1(ch1,ch2[0:p-1])
      if ch1[n-1] == ch2[p-1]
         val3 = lev1(ch1[0:n-1],ch2[0:p-1])
      else :
         val3 = 1 + lev1(ch1[0:n-1],ch2[0:p-1])
      minimum = min(val1,val2,val3)
      return minimum
```

Malheureusement, une fois de plus, cet algorithme est très lent en raison du **chevauchement des sous-problèmes**. Des distances déjà calculées sont recalculées plusieurs fois. Par exemple `lev1("AGORRYTNES","ALGORITHMES")` provoque 27 711 949 appels récursifs, ce qui lui prend plusieurs dizaines de secondes !

On écrit donc une version récursive avec mémoïsation : elle utilise un dictionnaire dont les clés sont les tuples (i, j) qui sont les longueurs des préfixes $\text{pref}_{\text{ch}_1}(i)$ et $\text{pref}_{\text{ch}_1}(j)$ et les valeurs sont les distance $d(i, j)$

Algo2(ch$_1$, ch$_2$) : version récursive avec mémoïsation utilisant un dictionnaire.

```
dico = {}

def lev2(ch1,ch2) :
   n,p = len(ch1), len(ch2)
   if (n,p) in dico.keys() :
      return dico[(n,p)]
   elif n == 0 :
      dico[(n,p)] = p
      return p
   elif p == 0 :
      dico[(n,p)] = n
      return n
   else :
      val1 = 1 + lev2(ch1[0:n-1],ch2)
      val2 = 1 + lev2(ch1,ch2[0:p-1])
      if ch1[n-1] == ch2[p-1]
         val3 = lev2(ch1[0:n-1],ch2[0:p-1])
      else :
         val3 = 1 + lev2(ch1[0:n-1],ch2[0:p-1])
      minimum = min(val1,val2,val3)
      dico[(n,p)] = minimum
      return minimum
```

γ) Construction de la solution

On souhaite maintenant non seulement savoir quelle est la distance de Levenshtein entre les deux chaînes ch$_1$ et ch$_2$ mais aussi el nombre d'opérations d'insertions, suppression et/ou remplacement qu'il a fallu faire pour passer de ch$_1$ à ch$_2$.

Pour y arriver, il est nécessaire d'avoir des informations supplémentaires. Nous souhaitons donc que chaque fonction `lev2(ch1,ch2)` renvoie non seulement $d(n, p)$, mais aussi l'opération qu'elle a dû faire.

Une idée est qu'elle renvoie un tuple composé de trois éléments : `(d(n,p), Op)` où Op est une chaîne de caractères qui peut valoir `"Supp"`, `"Ins"`, `"Remp"` ou `"Rien"`.

Bien entendu, dans la version récursive avec mémoïsation il faut que le dictionnaire auxiliaire contienne ces valeurs : `dico` devient donc maintenant un dictionnaire dont les clés sont (n, p) et dont les valeurs sont $(d(n, p), Op)$.

```
dico = {}

def lev2_modif(ch1,ch2) :
   n,p = len(ch1), len(ch2)
   if (n,p) in dico.keys() :
      return dico[(n,p)]
   elif n == 0 :
      dico[(n,p)] = (p,"Ins")
      return (p,"Ins")
   elif p == 0 :
      dico[(n,p)] = (n, "Supp")
      return (n,"Supp")
   else :
      val1 = 1 + lev2_modif(ch1[0:n-1],ch2)[0]
      val2 = 1 + lev2_modif(ch1,ch2[0:p-1])[0]
      if ch1[n-1] == ch2[p-1]
          val3 = lev2_modif(ch1[0:n-1],ch2[0:p-1])[0]
      else :
          val3 = 1 + lev2_modif(ch1[0:n-1],ch2[0:p-1])[0]
      minimum = min(val1,val2,val3)
      if minimum == val1 :
         resultat = (val1,"Ins")
      elif minimum == val2 :
         résultat = (val2,"Supp")
      else :
         if ch1[n-1] == ch1[p-1] :
            resultat = (val3, "Rien")
         else :
            resultat = (val3, "Remp")
      dico[(n,p)] = resultat
      return resultat
```

On reconstruit la solution sous la forme d'une liste dont les éléments sont "Ins", "Supp", "Remp" et "Rien"

```
dico = {}

def construire(ch1,ch2) :
   n,p = len(ch1),len(ch2)
   if n == 0 :
      return p*["Ins"]
   elif p == 0 :
      return n*["Supp"]
   else :
      mini, ch = lev2_modif(ch1,ch2)
      if ch == "Ins" :
         return ["Ins"] + construire(ch1,ch2[0:p-1])
      elif ch == "Supp" :
         return ["Supp"] + construire(ch1[0:n-1],ch2)
      elif ch == "Remp" :
         return ["Remp"] + construire(ch1[0:n-1],ch2[0:p-1])
      else :
         return ["Rien"] + construire(ch1[0:n-1],ch2[0:p-1])
```

Exemple :

`construire("AGORRYTNES","ALGORITHMES")` renvoie :
`["Rien", "Rien","Ins","Remp","Rien","Supp","Remp","Rien","Rien","Rien","Ins","Rien"]`

Il ne reste plus qu'à présenter les résultats de façon plus jolie, par exemple en les rassemblant dans un dictionnaire `bilan` don les clés sont `"Ins"`, `"Supp"` et `"Remp"` (on ne tient pas compte des "Rien" qui au fond ne nous intéressent pas). Voici un exemple d'encodage dans une fonction `presentation` :

```
dico = {}

def construire(ch1,ch2) :
   ... # rédaction du corps de la fonction construire

def presentation(ch1,ch2) :
   bilan = {}
   L = construire(ch1,ch2)
   for ch in L :
      if ch != "Rien" :
         if ch in bilan.keys() :
            bilan[ch] += 1
         else :
            bilan[ch] = 1
   return bilan
```

`presentation("AGORRYTNES","ALGORITHMES")` renvoie `{"Ins":2, "Remp":2, "Supp":1}`

3) Résumé

La programmation dynamique est une technique qui peut s'appliquer lorsque :

1. La solution optimale d'un problème P peut s'exprimer en fonction des solutions optimales de r sous-problèmes P_1, ..., P_r, via une sorte de *relation de récurrence*. On dit que le problème P possède une **propriété de sous-structure optimale**.
2. Il a des *redondances* lors de la résolution des sous-problèmes : la solution d'un sous-problème donné peut être recalculée plusieurs fois, ce qui augmente très rapidement la complexité d'un programme écrit de manière récursive. On dit qu'il y a **chevauchement** des sous-problèmes.

   Une technique de mémoïsation doit alors être appliquée aux programmes récursifs pour limiter leur complexité temporelle.

Quelle est la différence entre la programmation dynamique et la technique diviser pour régner que l'on applique par exemple dans le tri fusion ou le tri rapide ?

III. Quelques propriétés de suites récurrentes

Nous souhaitons étudier ici les suites récurrentes de réels $(u_n)_{n\in\mathbb{N}}$ vérifiant le relation de récurrence :

$$\forall\, n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n \qquad (1)$$

La suite de Fibonacci en est un exemple parmi d'autre puisqu'elle correspond à $u_0 = 0$ et $u_1 = 1$. En prenant d'autres valeurs initiales u_0 et u_1 on obtiendra d'autres suites. Nous noterons E l'ensemble des suites vérifiant (1).

On sait qu'une suite de réels est une application de $\mathbb{N}$ dans $\mathbb{R}$, c'est à dire un élément de $\mathscr{F}(\mathbb{N},\mathbb{R})$. Afin de simplifier l'écriture des théorèmes, nous utiliserons la notation u (comme une application) pour désigner une suite :

$$u : \mathbb{N} \longrightarrow \mathbb{R}, \quad n \longmapsto u(n)$$

et nous désignerons l'image de l'entier n par $u(n)$ au lieu de u_n.

Nous pouvons munir $\mathscr{F}(\mathbb{N},\mathbb{R})$ d'un loi interne additive + et d'une loi externe . à domaines d'opérateurs $\mathbb{R}$, définies par :

- Pour tout $(u, v) \in \mathscr{F}(\mathbb{N},\mathbb{R}) \times \mathscr{F}(\mathbb{N},\mathbb{R})$:

$$u + v : n \longmapsto u(n) + v(n)$$

- Pour tout $(\alpha, u) \in \mathbb{R} \times \mathscr{F}(\mathbb{N},\mathbb{R})$:

$$\alpha.u : n \longmapsto \alpha\, u(n)$$

On montre alors facilement que le triplet $(\mathscr{F}(\mathbb{N},\mathbb{R}), +, .)$ est un $\mathbb{R}$-espace vectoriel.

Proposition

$(E, +, .)$ est un sous-espace vectoriel de $\mathscr{F}(\mathbb{N},\mathbb{R})$.

Preuve :

Il suffit de montrer que E n'est pas vide et qu'il est stable pour les deux lois + et .

- E n'est pas vide puisque la suite nulle en est un élément. La suite de Fibonacci en consitue d'ailleurs un autre élément.
- Soient $(u, v) \in E^2$ et $(\alpha, \beta) \in \mathbb{R}^2$. Nous avons donc, pour tout $n \in \mathbb{N}$ et en posant $w = \alpha.u + \beta.v$:

$$\begin{aligned} w(n+2) &= \alpha\, u(n+2) + \beta\, v(n+2) \\ &= \alpha\,(\, u(n+1) + u(n)\,) + \beta\,(\, v(n+1) + v(n)\,) \\ &= (\alpha\, u(n+1) + \beta\, v(n+1)) + (\alpha\, u(n) + \beta\, v(n)) \\ &= w(n+1) + w(n) \end{aligned}$$

ce qui montre que la suite w est un élément de E.

CQFD.

Nous allons maintenant montrer que E est de dimension 2 en explicitant une base de E.

Soit $u : n \longmapsto r$ une suite constante de valeur $r \neq 0$. Cherchons à quelle condition cette suite est un élément de E. Nous avons donc l'équation :

$$\forall\, n \in \mathbb{N},\ r^{n+2} = r^{n+1} + r^n \iff r^2 - r - 1 = 0$$

puisque $r \neq 0$. Comme le discriminant de cette équation est $\Delta = 5 > 0$, elle admet deux racines réelles :

$$\varphi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \varphi' = \frac{1-\sqrt{5}}{2}$$

φ est le *nombre d'or* et φ' vérifie :

$$\varphi' = \frac{1-\sqrt{5}}{2} = \frac{(1-\sqrt{5})(1+\sqrt{5})}{2\,(1+\sqrt{5})} = -\,\frac{2}{1+\sqrt{5}} = -\,\frac{1}{\varphi}$$

Montrons que les deux suites $u_\varphi = (\varphi^n)_{n\in\mathbb{N}}$ et $u_{\varphi'} = (\varphi'^n)_{n\in\mathbb{N}}$ forment une base de E.

- *Montrons qu'il s'agit d'un système libre.*

  Soit $(\alpha, \beta) \in \mathbb{R}^2$ tel que $\alpha.u_\varphi + \beta.u_{\varphi'}$. Pour tout $n \in \mathbb{N}$ nous avons :

  $$\alpha\, u_\varphi(n) + \beta\, u_{\varphi'}(n) = 0 \iff \alpha\,\varphi^n + \beta\,\varphi'^n = 0$$

  Comme cette équation doit être vérifiée en particulier pour $n = 0$ et $n = 1$, nous obtenons le système :

  $$\left\{\begin{array}{lcl} \alpha+\beta & = & 0 \\ \alpha\,\varphi+\beta\,\varphi' & = & 0 \end{array}\right. \quad \text{d'où} \quad \left\{\begin{array}{lcl} \beta & = & -\,\alpha \\ \alpha\,(\varphi-\varphi') & = & 0 \end{array}\right.$$

  ce qui montre que $\alpha = 0$ et $\beta = 0$ puique $\varphi - \varphi' = \sqrt{5} \neq 0$.

- *Montrons qu'il s'agit d'un système générateur.*

  Considérons $u \in E$ et formons la suite $v = u - \alpha.u_\varphi - \beta.u_{\varphi'}$ en choisissant α et β de sorte que :

  $$v(0) = u(0) - \alpha\, u_\varphi(0) - \beta\, u_{\varphi'}(0) = 0 \quad \text{et} \quad v(1) = u(1) - \alpha\, u_\varphi(1) - \beta\, u_{\varphi'}(1) = 0$$

  Cela est possible puisque si nous notons $u_0 = u(0)$ et $u_1 = u(1)$ (retour aux notations traditionnelles), α et β vérifient les équations :

  $$\left\{\begin{array}{lcl} \alpha+\beta & = & u_0 \\ \alpha\,\varphi+\beta\,\varphi' & = & u_1 \end{array}\right. \quad \text{ce qui donne} \quad \left\{\begin{array}{lcl} \alpha & = & \dfrac{u_0\,\varphi' - u_1}{\varphi'-\varphi} = \dfrac{u_1 - u_0\,\varphi'}{\sqrt{5}} \\ \\ \beta & = & \dfrac{u_0\,\varphi - u_1}{\varphi-\varphi'} = \dfrac{u_0\,\varphi - u_1}{\sqrt{5}} \end{array}\right. \qquad (2)$$

  Comme $v \in E$ (puisque c'est un espace vectoriel) v obéit à l'équation (1) et une récurrence simple sur n montre alors que, pour tout $n \in \mathbb{N},\ v(n) = 0$. Ainsi, v est la suite nulle et il en résulte que :

  $$u = \alpha.u_\varphi + \beta.u_{\varphi'}$$

  CQFD.

En conclusion, tout suite $u \in E$ s'écrit de façon unique sous la forme $u = \alpha.u_\varphi + \beta.u_{\varphi'}$, ce qui signifie que (en revenant aux notations traditionnelles) :

$$\boxed{\forall\, n \in \mathbb{R},\ u_n = \alpha\,\varphi^n + \beta\,\varphi'^n = \alpha\,\varphi^n + (-1)^n\,\frac{\beta}{\varphi^n}}$$

les deux coefficients réels α et β se calculant grâce aux équations (2), c'est à dire à partir de u_0 et u_1.

Exemple :

Dans le cas particulier de la suite de Fibonacci, nous avons $u_0 = 0$ et $u_1 = 1$, ce qui entraîne :

$$\forall\, n \in \mathbb{R},\ u_n = \frac{1}{\sqrt{5}} \left(\varphi^n - (-1)^n \frac{1}{\varphi^n} \right)$$

Application :

Considérons la version récursive naïve de l'algorithme qui calcule les valeurs de la suite de Fibonacci. Nous le re-écrivons ci-dessous :

```
def Fibo(n) :
   if n == 0 or n == 1 :
      return n
   else :
      val = Fibo(n-1) + Fibo(n-2)
   return val
```

Soit $\mathrm{Nb}(n)$ le nombre d'appels d'une fonction $\mathrm{Fibo}(i)$ en commençant par appeler $\mathrm{Fibo}(n)$. Nous avons l'équation :

$$\forall\, n \geqslant 2,\ \mathrm{Nb}(n) = 1 + \mathrm{Nb}(n-1) + \mathrm{Nb}(n-2) \quad \text{et} \quad \mathrm{Nb}(0) = 1 \ ;\ \mathrm{Nb}(1) = 1$$

le 1 étant présent pour comptabiliser l'appel de $\mathrm{Fibo}(n)$. En posant $p = n - 2$, nous pouvons écrire l'équation de récurrence précédente sous la forme :

$$\forall\, p \in \mathbb{N},\ \mathrm{Nb}(p+2) = 1 + \mathrm{Nb}(p+1) + \mathrm{Nb}(p) \ \ (3) \text{ avec } \ \mathrm{Nb}(0) = 1 \ \text{ et } \ \mathrm{Nb}(1) = 1$$

Une solution constante de (3) est -1, ce qui conduit à poser $\mathrm{Nb}(p) = C(p) - 1$. La suite C satisfait donc à la récurrence :

$$\forall\, p \in \mathbb{N},\ \mathrm{C}(p+2) = \mathrm{C}(p+1) + \mathrm{C}(p) \quad \text{avec} \quad \mathrm{C}(0) = 2 \quad \text{et} \quad \mathrm{C}(1) = 2$$

On peut donc en conclure que $C \in E$, où E est l'espace vectoriel étudié précédemment, ce qui conduit à écrire :

$$\begin{aligned} \forall\, n \in \mathbb{N},\ C(n) &= \frac{2 - 2\varphi'}{\sqrt{5}}\, \varphi^n + \frac{2\varphi - 2}{\sqrt{5}}\, \frac{(-1)^n}{\varphi^n} \\ &= \frac{1 + \sqrt{5}}{\sqrt{5}}\, \varphi^n + \frac{\sqrt{5} - 1}{\sqrt{5}}\, \frac{(-1)^n}{\varphi^n} \end{aligned}$$

Comme $\varphi > 1$, $(-1)^n/\varphi^n \longrightarrow 0$ si $n \longrightarrow +\infty$ et donc :

$$C(n) \underset{\infty}{\sim} \frac{1+\sqrt{5}}{\sqrt{5}}\, \varphi^n = \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1}$$

et, de même, comme $\mathrm{Nb}(n) \underset{\infty}{\sim} C(n)$, il vient :

$$\boxed{\mathrm{Nb}(n) \underset{\infty}{\sim} \frac{2}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{n+1}}$$