ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

# Algorithmique Avancée

## Traitement des chaînes de caractères

- Programmation Dynamique
- Distance de Levenshtein
- Compression de données


Animé par : Dr. ibrahim GUELZIM

Email : ib.guelzim@gmail.com

# Sommaire

- Rappels
  - Introduction et notions générales
  - Analyse et conception d'algorithmes
  - Complexité d'algorithmes classiques : 3 Tris de tableaux, 2 recherches dans un tableau, Schéma de Hörner
  - Preuves d'algorithmes
- Autres algorithmes de tri :
  - Tri par fusion
  - Tri par Tas
- Complexité moyenne :
  - Application au Tri rapide
  - Structures de Données Probabilistes :
    - Notions sur les Tables de Hachage et Fonctions de Hachage,
    - Bloom Filter,
    - Count Min Sketch
- Traitements de chaines de Caractères :
  - Recherche de motif dans une chaine de caractères
  - Programmation dynamique
  - Distance de Levenshtein
  - Compression de données

# Programmation Dynamique

- La programmation dynamique est une technique de programmation visant à donner les **solutions optimales** à un problème *P* .
- S'applique lorsque la résolution de *P* peut se faire en résolvant *r* sous-problèmes $P_1, \dots, P_r$ ,
  1. Commence par chercher les solutions optimales des sous-problèmes.
  2. Combine ces solutions optimales pour trouver les solutions optimales de *P* .
- Un programme dynamique :
  - Peut être rédigé en version itérative
  - En version récursive:
    - Utilisation de la technique de **la mémoïsation** par accélérer les calculs

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  - Calcul des termes de la suite de Fibonacci $(F_n)_{n \in N}$ en utilisant la définition suivante :
    - $F_0 = 0 \; ; \; F_1 = 1$ et $\forall n \geq 2, \; F_n = F_{n-1} + F_{n-2}$
  - Le problème P = "calculer Fn" peut être réalisé en cherchant la solution des deux sous-problèmes :
    - $P_1$ = "calculer $F_{n-1}$" et
    - $P_2$ = "calculer $F_{n-2}$".

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  a) Méthode itérative
     - <u>Algo1</u> : ( Python )

```
def Fibo1(n) :
        if n == 0 or n == 1 :
                return n
        else :
                T = (n+1)*[0]  # créer liste T de n + 1 entiers  pour stocker les Fi de 0 à n
                T[1] = 1
                for i in range(2,n+1) :
                        T[i] = T[i-1] + T[i-2]
        return T[n]
```

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  a) Méthode itérative
     - <u>Algo2</u> : version avec un dictionnaire ( Python )

```
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]
```

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  - b) Méthode récursive
    - <u>Algo3</u> : version naïve ( Python )

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

- Nombre d'appel à la récursivité Nb est exponentionnel :

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

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  - b) Méthode récursive
    - Algo3 : version naïve ( Python )
      - Pb : le même calcul est refait inutilement : illustration pour n = 5

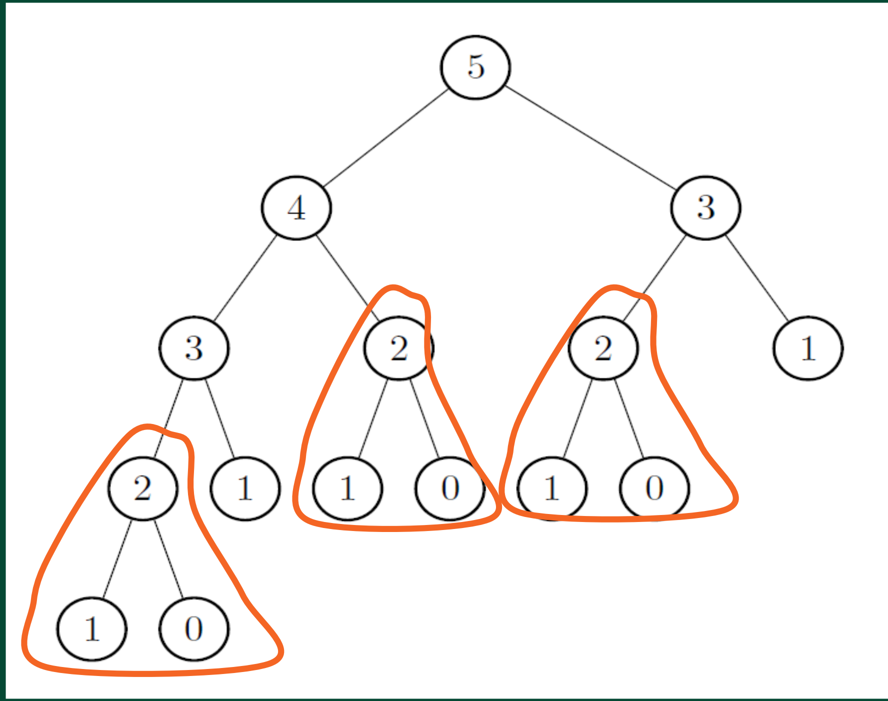
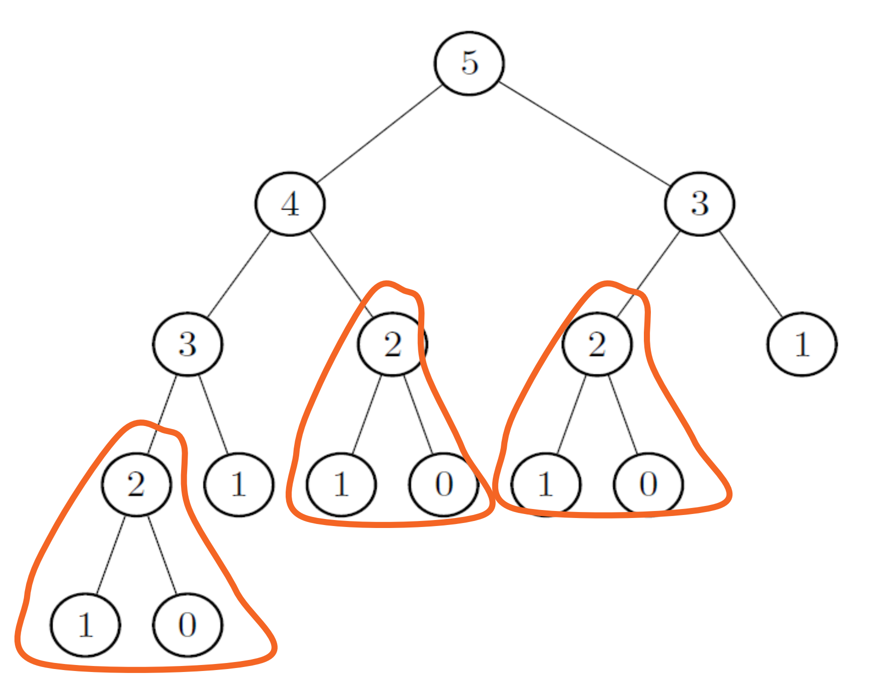



Fig 5.1. Arbre des appels récursifs pour `FiboRN`(5):
`FiboRN`(3) est appelé deux fois et `FiboRN`(2) trois fois

# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  - b) Méthode récursive
    - <u>Algo3</u> : version naïve ( Python )
      - Pb :
        - le même calcul est refait inutilement : illustration pour n = 5
        - Chevauchement des sous-problèmes.
      - Sol :
        - Stocker les résultats intermédiaires dans un tableau ( ou un dictionnaire ),

        → Éviter de recalculer ,
        - Opération appelée : mémoïsation .



# Technique de mémoïsation

- Illustration : Suite de Fibonacci
  - b) Méthode récursive
    - <u>Algo4</u> : version récursive avec mémoïsation

```
dicRM = {} # dictionnaire vide, défini comme variable globale
def FiboRMem(n)  :
        if n in dicRM.keys()  :
                return dicRM[n]
        elif n == 0 or n == 1 :
                dicRM[n]  = n
                return n
        else :
                val = FiboRMem( n-1 ) + FiboRMem( n-2 )
                dicRM[n]  = val
        return val
```

- Remarque :
  - `FiboRN(20)` fait 21 930 appels récursifs tandis que `FiboRMem(20)` n'en provoque que 39

## Problème du Rendu de Monnaie

- S'intéresse à des problèmes d'optimisation ,
- Appliquée à un problème P qui peut être résolu en commençant par résoudre r sous-problèmes $P_1$, ..., $P_r$ ,
- Illustration: Problème du rendu de monnaie
- On dispose de pièces de monnaie et de billets, par exemple des pièces de 1, 2, 5 et 10 Dirhams ( Dh ), puis des billets de 20, 50, 100 et 200 Dh.
- Nous appellerons indifféremment "pièces" les pièces de monnaie et les billets.
- On souhaite :
  - Rendre une somme d'argent *S* avec ces pièces
  - En utilisant le moins de pièces possible,
  - Sachant qu'on n'est pas limité et on dispose de toutes les pièces nécessaires.

# Problème du Rendu de Monnaie

- Formalisme mathématique :
  - Il y a 8 types de pièces que nous numérotons de 1 à 8 , et soient :
    - $m_i$ : le montant de la pièce de type n° $i$. alors :

      $m_1 = 1,\ m_2 = 2,\ m_3 = 5,\ m_4 = 10,\ m_5 = 20,\ m_6 = 50,\ m_7 = 100,\ m_8 = 200$

    - $n_i$ : le nombre de pièces de montant $m_i$ dans la somme S à rendre

    - n : le nombre de pièces rendues, nous avons :

      $$S = \sum_{i=1}^{8} n_i m_i \quad (1) \quad et \quad n = \sum_{i=1}^{8} n_i \ (2)$$

    - E(S) : l'ensemble des solutions de (1) pour une somme S à rendre :

      $$E(S) = \{ \ (n_1, n_2, \dots, n_8) \in \mathbb{N}^8 ) \mid S = \sum_{i=1}^{8} n_i m_i \}$$

    - $V(S) = \{ \ n = \sum_{i=1}^{8} n_i \ \mid (n_1, n_2, \dots, n_8) \in E(S) \} \subset \mathbb{N}$ ,

    - V(S) admet un plus petit élément noté $n_{min}(S)$

# Problème du Rendu de Monnaie

- Formalisme mathématique :
  - Soit : $E^*(S) = \{ ( n_1, n_2, \ldots , n_8 ) \in E(S) \mid \sum_{i=1}^{8} n_i = n_{\min}(S) \}$

    Ou

    $E^*(S) = \{ ( n_1, n_2, \ldots , n_8 ) \in E(S) \mid S = \sum_{i=1}^{8} n_i m_i \text{ et } \sum_{i=1}^{8} n_i = n_{\min}(S) \}$

  - $E^*(s)$ 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).

# Problème du Rendu de Monnaie ( PRM )

- Résolution par algorithme glouton (En : Greedy algorithm) :
  - Un algorithme glouton résout le problème étape par étape :
    - À chaque étape, donner une solution optimale ( optimum local )
    - Évaluer le reste à résoudre et passer à l'étape suivante.
    - Les solutions données aux étapes précédentes ne sont jamais remises en question.
  - Solution PRM :
    1. Prendre la pièce de plus grande valeur m inférieure à s.
    2. Rendre le nombre maximal n de pièces de valeur m tel que $nm \leq s$.
    3. Calculer ce qui reste à rendre : $s - nm$.
    4. Recommencer l'étape 1 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).

# Problème du Rendu de Monnaie ( PRM )

- Résolution par algorithme glouton (En : Greedy algorithm) :
  - Solution PRM : Implémentation
    - Utiliser un dictionnaire pour stocker la valeur et le nombre de pièces rendues.
    - Formé des couples (m : n) (les clés sont les montants m de chaque pièce).

```
PieceMonnaie = [200, 100, 50, 20, 10, 5, 2, 1]
Rendu = {}
s = 950
for m in PieceMonnaie :
    if m <= s :
        n = s // m
        Rendu[m] = n
        s = s % m
n_min = 0
for m in Rendu.keys() :
   n_min += Rendu[m]
print("Rendu :",Rendu)
print("nbr piece : ",n_min)
```

# Problème du Rendu de Monnaie ( PRM )

- Résolution par algorithme glouton (En : Greedy algorithm) :
  - Q : Solution de l'algorithme glouton est optimale ?
  - R : *Cela dépend du jeu de pièces de monnaie qu'on possède.*
  - Exemple:
    - Les pièces de monnaies disponibles : [ 7, 5, 1 ].
    - Que renvoie l'algorithme glouton si $s = 11$ ?
      - Rendu : {7: 1,  1: 4}
      - nbr pièces :  5
    - Est elle la solution optimale ?
      - Non :
        - Rendu : {5: 2,  1: 1}
        - nbr pièces :  3

# Problème du Rendu de Monnaie ( PRM )

- Résolution par programmation dynamique :
  - Le problème est le suivant :
    - Pour une somme S donnée, quel est le nombre minimal $n_{min}(S)$ de pièces à rendre, càd le plus petit élément de V (s) :

$$V(S) = \{ \ n = \sum_{i=1}^{8} n_i \ \mid ( n_1, n_2, \dots , n_8 ) \in \mathbb{N} \text{ et } S = \sum_{i=1}^{8} n_i m_i \},$$

  - On peut remarquer que :
    - $V( 0 ) = \{ 0 \}$
    - Pour rendre la somme S, il faut commencer par rendre une pièce :
      - si on choisit une pièce d'un montant $m_k \leq s$,
      - il reste à rendre la somme $S - m_k < S$, d'où :

$$V(S) = \bigcup_{\substack{1 \leq k \leq 8 \\ m_k \leq s}} \{ \ 1 + n \mid n \in V( S - mk ) \ \}$$

      - Et donc :

$$\forall \ S > 0, n_{min}( S ) = \min_{\substack{1 \leq k \leq 8 \\ m_k \leq s}} ( 1 + n_{min}( S - m_k ) ) \text{ et } n_{min} ( 0 ) = 0 \qquad (*)$$

# Problème du Rendu de Monnaie ( PRM )

- On pourra utiliser la propriété suivante :
  - Si $A$ et $B$ sont deux parties non vides de N et que $a = \min(A)$ et $b = \min(B)$ alors : $\min(A \cup B) = \min(a, b)$
  - Généralisable par récurrence à toute réunion **finie** de parties de 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 $n_{min}(s - m_k)$ ;

- 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.

# Problème du Rendu de Monnaie ( PRM )

- Algo1(s) : version récursive naïve
  - Implémentation Python :

```
Monnaie = [1,2,5,10,20,50,100,200]
def PRM_recN(s) :
    if s == 0 :
            return 0
    else :
            L = []
            for m in Monnaie :
                    if m <= s :
                            N = 1 + PRM_recN( s - m )
                            L.append(N)
                    else :
                            break
    val = min(L)
    return val
```

# Problème du Rendu de Monnaie ( PRM )

- Remarque / Algo1 ( Rec Naïf )
  - Solution gourmande,
  - Ex:

Arbre des appels récursifs,

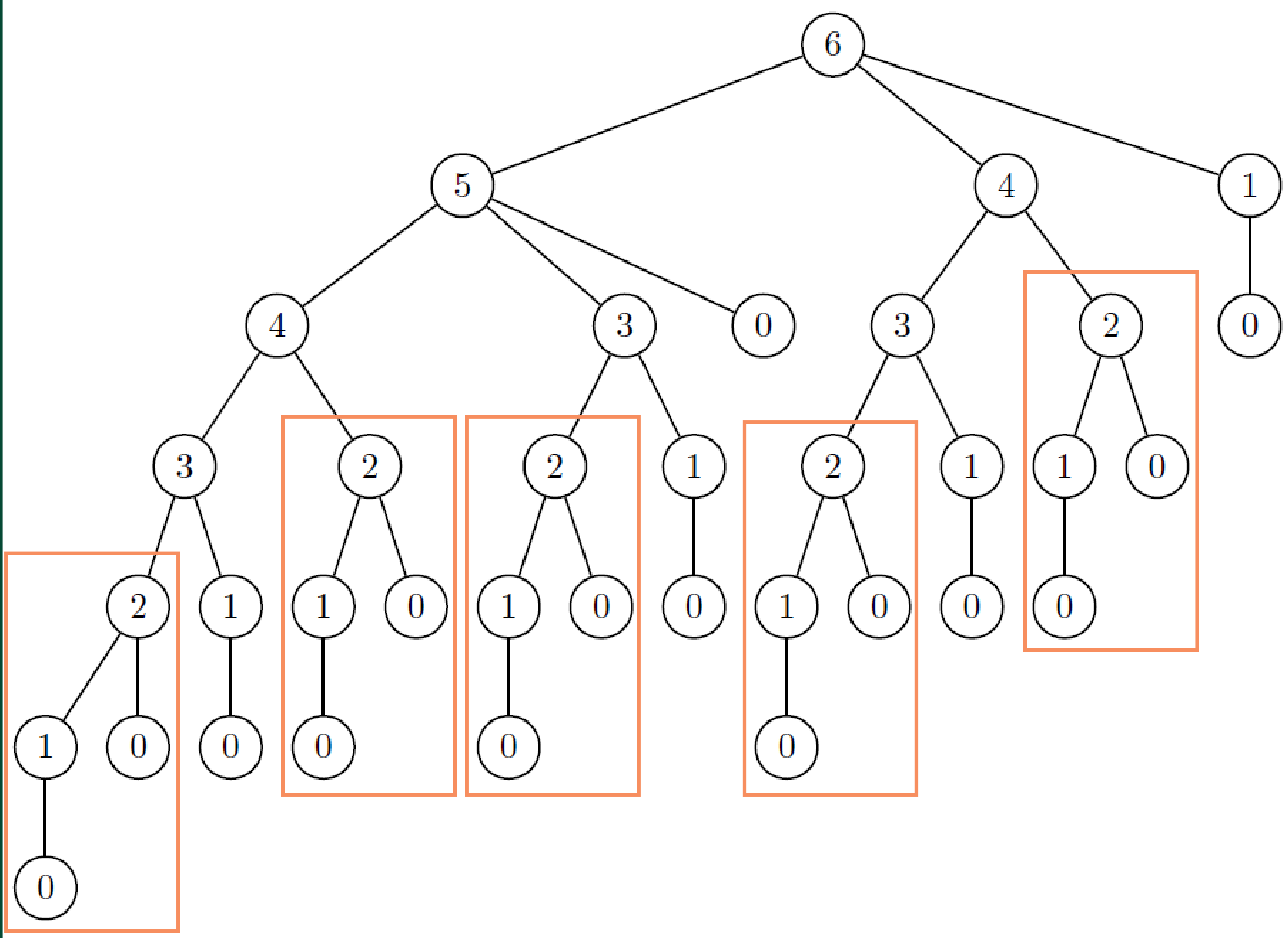
pour : `PRM_recN(6)`

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

- Il y a donc un **chevauchement** des sous-problèmes

→ Technique de mémoïsation nécessaire pour accélérer les calculs.

→ Utiliser un dictionnaire dont les éléments seront les couples ($s$ : $n\_min(s)$) pour stocker les résultats et ne pas refaire du calcul déjà fait.



# Problème du Rendu de Monnaie ( PRM )

- *Algo2 Rec avec* *__Memoïsation__* *( utilisation de dictionnaire ) :*

```
Monnaie = [1,2,5,10,20,50,100,200]; dico = {}
def PRM_recD(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 :
                N = 1 + PRM_recD( s - m )
                L.append(N)
            else :
                break
        val = min(L)
        dico[s] = val
        return val
```

# Problème du Rendu de Monnaie ( PRM )

- *Algo3* : itératif

```
Monnaie = [1,2,5,10,20,50,100,200]
def PRM_iter(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]
```

# Problème du Rendu de Monnaie ( PRM )

- *ALgo 4: itératif avec dictionnaire*

```
Monnaie = [1,2,5,10,20,50,100,200]
def PRM_iterD(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]
```

## Distance de Levenshtein ( https://fr.wikipedia.org/wiki/Distance_de_Levenshtein )

- Mesure la différence entre deux chaînes de caractères.

- Égale au nombre minimal d'opérations nécessaires pour transformer une chaîne de caractères en une autre, à l'aide des trois opérations autorisées :
  - Substitution ( ou remplacement ),
  - Suppression,
  - Insertion,

- Appelée aussi distance d'édition

# Distance de Levenshtein

- Supposons que nous avons deux chaines A de longueur i et B de longueur j
- Supposons qu'on a aligné A[ 0 : i-1 ] et B[ 0 : j-1 ]
- On ajoute un caractère x à la fin de la chaine A et y à la fin de B,
- Pour aligner de nouveau A et B on est devant 4 cas d'opérations sur A :
    - x = y : ne rien faire
    - x ≠ y : remplacer x par y
    - x = ε (vide) : insérer y
    - y = ε (vide) : supprimer x

- Le coût du 1$^{er}$ cas est : 0, tandis que le coût des 3 dernières opérations est égale à 1.

# Distance de Levenshtein

- Soient deux chaînes a et b,
- telles que |a| est le cardinal de a ou son nombre de lettres, et
- a–1 est la chaîne a tronquée de sa 1ère lettre a[0]
- Formellement, la distance de Levenshtein, entre a et b est :

$$
\text{lev}(a,b) = \begin{cases} \max(|a|, |b|) & \text{si } \min(|a|, |b|) = 0 \\ \text{lev}(a-1, b-1) & \text{si } a[0] = b[0] \\ 1 + \min(\text{lev}(a-1, b), \text{lev}(a, b-1), \text{lev}(a-1, b-1)) & \text{sinon} \end{cases}
$$

# Distance de Levenshtein

- Solution 1:

```
def levRec(a,b):
    if len(a) == 0:
        return len(b)
    elif len(b) == 0:
        return len(a)
    elif a[0] == b[0]:
        return levRec(a[1:],b[1:])
    else:
        return 1 +
        min(levRec(a[1:],b),
        levRec(a,b[1:]),
        levRec(a[1:],b[1:]))
```

# Distance de Levenshtein

- **Solution 2:**

```
def lev_distance (a, b):
    # Création de la matrice de distance
    la = len(a) ;    lb = len(b)
    DL = [[0 for x in range(lb+1)] for y in range(la+1)]

    # Initialisation de la première ligne et  première colonne de la matrice
    for i in range(la+1):
        DL[i][0] = i
    for j in range(lb+1):
        DL[0][j] = j

    # Remplissage de la matrice de Distance DL
    # DL[i][j] représente la distance entre a[:i] et b[:j]
    for i in range(1, la+1):
        for j in range(1, lb+1):
            cout = (a[i - 1] != b[j - 1])
            DL[i][j] = min(   DL[i - 1][j] + 1,            # Suppression
                              DL[i][j - 1] + 1,            # Insertion
                              DL[i - 1][j - 1] + cout)     # Substitution
    return DL[la][lb]
```

# Distance de Levenshtein

- Exemple : A = "NICHE" et B = "CHIENS"
  - La matrice DL fournit les suites d'opérations possibles ( ici en nombre de 6 )
  - On part d'en bas à droite DL[la][lb] vers DL[0][0]
  - On examine les trois cases du quartier supérieur gauche ; (4,4,5) dans le sens d'une montre.
  - On suit les valeurs minimums.
  - Ici il y a deux 4 qui donnent deux chemins (une bifurcation), et ainsi de suite on crée un arbre par récurrence.
  - Interprétation des actions :
    - *Aller au-dessus* : *détruire la lettre de la ligne*;
    - *Aller en diagonale* : *substituer la lettre de la ligne par la lettre de la colonne* (ou pas si elles sont égales) ;
    - *Aller à gauche* : *Ajouter la lettre de la colonne*.
  - À la fin, il faut lire les étapes à l'envers.

| | | C | H | I | E | N | S |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| N | 1 | 1 | 2 | 3 | 4 | 4 | 5 |
| I | 2 | 2 | 2 | 2 | 3 | 4 | 5 |
| C | 3 | 2 | 3 | 3 | 3 | 4 | 5 |
| H | 4 | 3 | 2 | 3 | 4 | 4 | 5 |
| E | 5 | 4 | 3 | 3 | 3 | 4 | 5 |

# Distance de Levenshtein

- **Applications de la distance de Levenshtein**
  - **Vérification orthographique:** Pour suggérer des corrections d'orthographe.
  - **Recherche de similarité de chaînes de caractères:** Pour trouver des chaînes similaires dans une base de données.
  - **Bioinformatique:** Pour comparer des séquences d'ADN ou de protéines (alignement).
  - **Reconnaissance vocale:** Pour corriger les erreurs de transcription.
  - **Traitement du langage naturel:** Pour la segmentation de mots, la correction automatique, etc.

## References


- Introduction à l’algorithmique. Cours et exercices. Cormen et al. 2e édition. (En 3rd Edition)
- https://cahier-de-prepa.fr/mp1-janson/download?id=2654
- Algorithms, FOURTH EDITION, Robert Sedgewick and Kevin Wayne. Princeton University.
- https://fr.wikipedia.org/wiki/Distance_de_Levenshtein
- https://jeffe.cs.illinois.edu/teaching/algorithms/