Programmation dynamique - EXERCICES

Exercice 1 : Rendu de monnaie

Version gloutonne (rappel)

Vous avez étudié ce problème du rendu de monnaie en classe de Première : il faut rendre une somme `s` avec le moins de pièces possibles. Nous avions résolu ce problème grâce à un **algorithme glouton** qui consistait à :

1. Prendre la pièce de plus grande valeur inférieure ou égale à la somme restant à rendre
2. Recommencer l'opération 1 jusqu'au moment où la somme à rendre est égale à 0.

Pour rappel, voilà une fonction réalisant cet algorithme glouton :

In [ ]:
```
def rendu_monnaie_glouton(s, pieces):
        """Renvoie la solution gloutonne du rendu de monnaie d'une somme s entière e
        Le tableau pieces contient les valeurs des pièces à disposition dans l'ordre
        solution = []
        i = len(pieces) - 1 # position de la première pièce à tester (la plus grande
        while s > 0 and i >= 0: # tant qu'il reste de l'argent à rendre et que toute
            valeur = pieces[i] # on prend la pièce d'indice i
            if valeur <= s: # s'il est possible de rendre la pièce
                solution.append(valeur) # on l'ajoute à solution
                s = s - valeur # et on déduit sa valeur de la somme à rendre
            else:
                i = i - 1 # sinon on passe à la pièce immédiatement inférieure
        return solution

# ESSAI

euros = [1, 2, 5, 10, 20, 50, 100, 200, 500]
rendu_monnaie_glouton(147, euros)
```

Question 1 : Appliquez l'algorithme glouton dans les différents cas suivants :

1. s = 27 et pieces = [1, 2, 5, 10, 20, 50, 100, 200, 500]
2. s = 8 et pieces = [1, 4, 6]
3. s = 8 et pieces = [5, 2]

Question 2 : Quelle analyse pouvez-vous faire de ces solutions gloutonnes ?

Question 3 : Modifiez la fonction précédente pour qu'elle renvoie le nombre de pièces à utiliser avec une résolution gloutonne (et non plus une liste des pièces).

In [ ]:
```
# à vous de jouer !
```

Version récursive naïve

Dans cette partie, on veut écrire un autre algorithme de rendu de monnaie qui donne toujours la solution optimale, quelles que soient les pièces à disposition (que l'on suppose en quantité illimité).

> **Important** : on supposera que l'on dispose toujours de la pièce unité, de sorte de pouvoir rendre n'importe quelle somme, dans le pire des cas avec $1 + 1 + \ldots + 1$.

L'algorithme proposé dans ce qui suit consiste à chercher *toutes* les façons possibles de rendre la somme `s` . On va procéder de manière *récursive* en exploitant l'idée suivante : pour rendre une somme `s` de façon optimale, il faut rendre une pièce `p` et la somme `s-p` de façon optimale. L'objectif est donc de trouver la pièce `p` idéale, autrement dit la valeur `p` qui minimise le nombre de pièces pour faire la somme `s-p` . Pour cela, on teste toutes les pièces `p` possibles et on calcule récursivement le nombre minimal de pièces pour faire la somme `s-p` .

Exemple

Dans la suite, on note $\mathrm{nb\_pieces}(s)$ le nombre minimal de pièces pour rendre la somme s.

Imaginons que l'on veuille rendre la somme 100 avec des pièces de 1 et 2. On cherche donc $\mathrm{nb\_pieces}(100)$. Il y a deux cas de figure :

- Soit on rend une pièce de 1 et il faut encore rendre (de façon optimale) 100 - 1 = 99.
- Soit on rend une pièce de 2 et il faut encore rendre (de façon optimale) 100 - 2 = 98.

Pour savoir quelle pièce rendre, il faut déterminer le nombre minimal de pièces pour rendre 99 (c'est-à-dire $\mathrm{nb\_pieces}(99)$) et pour rendre 98 (c'est-à-dire $\mathrm{nb\_pieces}(98)$) et choisir le minimum de deux possibilités. Il suffira d'ajouter 1 à ce nombre puisqu'on rend une pièce de plus (la pièce 1 ou la pièce 2).

Mais pour connaître $\mathrm{nb\_pieces}(99)$, il faut procéder de la même façon : soit on rend 1 et il faut trouver $\mathrm{nb\_pieces}(98)$, soit on rend 2 et il faut trouver $\mathrm{nb\_pieces}(97)$, puis ajouter 1 et prendre le minimum des deux cas de figure.

Puis pour trouver $\mathrm{nb\_pieces}(98)$, il faut faire deux appels récursifs à $\mathrm{nb\_pieces}(97)$ et $\mathrm{nb\_pieces}(96)$. Et ainsi de suite...

Question 4 : Dessinez les 4 premiers niveaux de l'arbre d'appels si on lance $\mathrm{nb\_pieces}(100)$ puis comptez le nombre d'appels pour chaque valeur de s. Souhaitez-vous construire l'abre d'appels complet ?

Implémentation

Question 5 : Complétez la définition suivante de la fonction $\mathrm{nb\_pieces}$ pour faire apparaître le cas de base et le cas récursif.

$$\mathrm{nb\_pieces}(s) = \begin{cases} \ldots & \text{si } s = 0 \\ \ldots & \text{si } \ldots \end{cases}$$

Question 6 : Complétez les `???` pour que la fonction récursive `rendu_monnaie_naif` qui suit réalise l'algorithme naïf du rendu de monnaie.

In [ ]:
```
# à vous de jouer !
def rendu_monnaie_naif(s, pieces):
    """Renvoie le nombre minimal de pièces pour rendre la somme s avec les pièces st
    if s == 0:
        return ???
    nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
    for p in pieces:
        if p <= s:   # inutile de tester une pièce dont la valeur dépasse la somme s
            nb_pieces_bis = 1 + rendu_monnaie_naif(???, ???)
            nb_pieces = min(???, ???)
    return nb_pieces
```

Question 7 : L'algorithme glouton ne donnait pas la solution optimale dans le cas où s = 8 et pieces = [1, 4, 6]. Vérifiez que l'on obtient désormais la solution optimale avec la version récursive naïve.

In [ ]:
```
# à vous de jouer !
```

Efficacité

Question 8 : Vérifiez que si on essaye de calculer `rendu_monnaie_naif(100, [1, 2])` , le programme ne termine pas !

In [ ]:
```
# à vous de jouer !
```

> L'appel en question génère plus de 1500 milliards de milliards d'appels récursifs 😳, ce qui montre que cette solution récursive naïve est beaucoup trop coûteuse.

Question 9 : Utilisez la commande magique `%time` pour évaluer le temps d'exécution du calcul `rendu_monnaie_naif(35, [1, 2])` .

In [ ]:
```
# à vous de jouer !
```

Pour terminer cette partie, on va chercher à évaluer le nombre d'appels à la fonction `rendu_monnaie_naif` pour chaque valeur de `s` possible lorsque l'on lance un appel sur une somme `s` à rendre.

Pour cela, on peut utiliser un dictionnaire, que l'on complète et met à jour au fur et à mesure des appels récursifs.

Question 10 : On a écrit une fonction `nombre_appels(s, pieces)` ci-dessous. Complétez la fonction `rendu_monnaie_naif_bis(s, pieces, dico)` qui renvoie toujours le nombre minimal de pièces pour rendre la somme `s` mais qui complète et met à jour également le dictionnaire `dico` avec le nombre d'appels intermédiaires pour chaque valeur de `s-p` .
Indication : il suffit d'ajouter quelques instructions à la fonction de la question 6.

In [ ]:
```
def nombre_appels(s, pieces):
    '''Renvoie un dictionnaire qui associe à chaque argument son nombre d'appels réc
    pour rendre la somme s avec le système pieces.

    >>> nombre_appels(8, [1, 4, 6])
    {8: 1, 7: 1, 6: 1, 5: 1, 4: 2, 3: 3, 2: 5, 1: 7, 0: 10}

    Interprétation : la fonction est appelée 1 fois avec l'argument 8, 1 fois avec l
    ..., 10 fois avec l'argument 0.
    '''
    N = {s: 1}  # on appelle une fois la fonction pour la somme s de départ
    rendu_monnaie_naif_bis(s, pieces, N)
    return N

def rendu_monnaie_naif_bis(s, pieces, dico):
    # à compléter
```

Question 11 :

1. Vérifiez que l'appel `nombre_appels(8, [1, 4, 6])` renvoie le bon dictionnaire.
2. Observez le nombre d'appels nécessaires à la fonction `rendu_monnaie_naif` pendant le calcul de `rendu_monnaie_naif(35, [1, 2])` ?

In [ ]:
```
# à vous de jouer !
```

In [ ]:
```
# à vous de jouer !
```

> On comprend aisément avec ces exemples, l'intérêt de ne pas calculer plusieurs fois la même chose. La suite est consacrée à l'amélioration de l'algorithme en utilisant la programmation dynamique selon une approche récursive avec mémoisation puis selon une approche itérative ascendante.

Version récursive avec *mémoisation*

Mémorisation dans un dictionnaire

Question 12 : En vous inspirant de ce qui a été vu en cours sur le calcul du terme de rang n de la suite de Fibonacci, proposez une fonction `rendu_monnaie_memo(s, pieces, memo)` qui renvoie le nombre minimal de pièces pour rendre la somme `s` et qui utilise un dictionnaire `memo` pour stocker les résultats déjà connus afin de ne pas les calculer plusieurs fois.

In [ ]:
```
# à vous de jouer !
```

Question 13 : Ecrivez une fonction d'interface `rendu_monnaie_memoisation(s, pieces)` qui renvoie le nombre minimal de pièces pour rendre la somme `s` avec le sytèmes `pieces`, en utilisant la fonction `rendu_monnaie_memo` . *Voir ce qui a été fait avec la suite de Fibonacci si besoin.*

In [ ]:
```
# à vous de jouer !
```

Question 14 : Vérifiez que l'on obtient beaucoup plus rapidement le résultat pour rendre la somme 35 avec les pièces `[1, 2]` et que l'algorithme renvoie une réponse (très rapide également) pour rendre la somme 100 avec les mêmes pièces

In [ ]:
```
# à vous de jouer !
```

Mémorisation dans un tableau

Question 15 : Modifiez les deux fonctions précédentes pour utiliser cette fois-ci un **tableau** `memo` pour stocker les valeurs déjà calculées. Dans ce tableau, l'élément en position i sera le nombre optimal de pièces pour rendre la somme i. On pourra par exemple initialiser le tableau avec toutes les valeurs à `None` .

In [ ]:
```
# à vous de jouer !
```

Efficacité

Question 16 : On lance l'appel `rendu_monnaie_memoisation(100, [1, 2])` .

1. Dénombrez le nombre d'appels récursifs *non mémoïsés* engendrés par cet appel.
2. Dans le cas d'un appel récursif non mémoïsé, combien de passage y a-t-il dans la boucle `for` ?

Question 17 : On considère que `memo[n]` est accessible en temps constant ($O(1)$). Dans le cas général, évaluez le coût en temps de l'algorithme avec mémoïsation. Et le coût en espace ?

Version itérative *ascendante*

Algorithme

On va utiliser un tableau `nb` dans lequel on va stocker successivement les différents nombres de pièces à rendre pour $n = 0$, $n = 1$, ... jusqu'à $n = s$.

Le tableau `nb` a donc pour taille $s + 1$ au départ, que l'on peut créer en le complétant avec des zéros pour commencer.

A la fin de l'algorithme, `nb[n]` (pour `n` de 0 à s) doit contenir le nombre de pièces optimal pour rendre la somme `n` .

La première case du tableau est correctement initialisée, dont il suffira de remplir le tableau pour toutes les sommes de 1 à s. Pour cela, on peut utiliser une boucle `for` qui parcoure les différentes sommes possibles par ordre croissant des indices (de 1 à s) en utilisant la même formule que pour la version récursive :

$$\mathrm{nb}[n] = 1 + \min_{p \leq n} (\mathrm{nb}[n - p])$$

Exemple

On souhaite faire tourner cet algorithme sur l'exemple $s = 8$ et $pieces = [1, 2, 5]$.

Le tableau `nb` est initialisé à `[0, 0, 0, 0, 0, 0, 0, 0, 0]` et on va le remplir indice par indice.

- Pour $n = 1$:
  - on initialise `nb[1]` à `1` puisque dans le pire des cas, on peut rendre la somme 1 avec une pièce ($1 = 1$)
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 0 et on sait que `nb[0] = 0` donc cela ferait $1 + 0 = 1$ pièce ✅
    - on ne peut pas rendre la pièce 2 ❌
    - on ne peut pas rendre la pièce 5 ❌
  - à la fin de l'itération, on a donc `nb[1] = 1` et donc `nb = [0, 1, 0, 0, 0, 0, 0, 0, 0]`
- Pour $n = 2$:
  - on initialise `nb[2]` à `2` puisque dans le pire des cas, on peut rendre la somme 1 avec 2 pièces de 1 ($2 = 1 + 1$)
  - on regarde ensuite si on peut faire mieux en analysant tous les cas possibles :
    - si on rend la pièce 1, il faut encore rendre 1 et on sait que `nb[1] = 1` donc cela ferait $1 + 1 = 2$ pièces ❌
    - si on rend la pièce 2, il faut encore rendre 0 et on sait que `nb[0] = 0` donc cela ferait $1 + 0 = 1$ pièce ✅
    - on ne peut pas rendre la pièce 5 ❌
  - à la fin de l'itération, on trouve que `nb[2] = 1` et donc `nb = [0, 1, 1, 0, 0, 0, 0, 0, 0]`
- Ainsi de suite, jusqu'à $n = 8$.

Question 18 : Poursuivez cette méthode en précisant toutes les étapes. Dans quelle case du tableau se trouve la réponse au problème ?

Implémentation

Question 19 : Ecrivez une fonction `rendu_monnaie_ascendante(s, pieces)` qui renvoie le nombre minimal de pièces pour rendre la somme `s` avec le système `pieces` en utilisant la méthode ascendante décrite juste au-dessus. *La recherche du minimum suit le même principe que dans les versions récursives précédentes.*

In [ ]:
```
# à vous de jouer !
```

Question 20 : Vérifiez que cette fonction renvoie la même réponse que ce que vous avez trouvé à la question précédente. Lancez quelques appels pour vérifier l'efficacité de cet algorithme de programmation dynamique.

In [ ]:
```
# à vous de jouer !
```

Construction d'une solution

Le programme précédent renvoie le nombre de pièces minimales mais on ne sait pas lesquelles pour autant. L'objectif est de modifier ce dernier pour qu'il renvoie une solution possible du problème.

Pour cela, il suffit de créer un deuxième tableau `sol` qui contient, pour chaque somme entre 0 et s, une solution minimale pour cette somme. Il suffira alors de remplir ce tableau à chaque fois que l'on remplit le tableau `nb` en ajoutant la nouvelle pièce `p` retenue à celles de la solution pour rendre `s-p` .

On peut initialiser `sol` par un tableau vide dans chaque case.

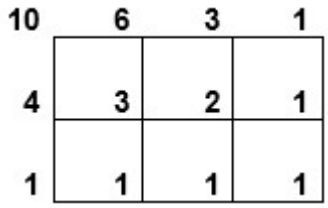
Question 21 : Ecrivez une fonction `rendu_monnaie_ascendante_solution(s, pieces)` qui renvoie une liste minimale de pièces permettant de rendre la somme `s` avec le système `pieces` . **Attention** : il faudra veiller à copier, avec la méthode `copy` , les listes solution avant d'ajouter la nouvelle pièce

In [ ]:
```
# à vous de jouer !
```

Exercice 2 : Retour sur l'exemple d'introduction du cours

On souhaite écrire une fonction `chemins(n, m)` qui calcule le nombre de chemins sur une grille $n \times m$, qui mènent du coin supérieur gauche, au coin inférieur droit, en se déplaçant uniquement horizontalement vers la droite et verticalement vers le bas.

Relation de récurrence



On rappelle que chaque nombre est la somme du nombre situé à droite et du nombre situé en-dessous. Autrement dit, on a la relation de récurrence suivante :

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

Version récursive naïve

La relation de récurrence permet d'écrire la fonction récursive naïve suivante :

In [ ]:
```
def chemins(n, m):
    if n == 0 or m == 0:
        return 1
    else:
        return chemins(n - 1, m) + chemins(n, m - 1)

chemins(3, 2), chemins(10, 10)
```

Vous pouvez constater que pour des valeurs un peu plus élevée, le calcul prend plusieurs secondes et si on augmente encore les valeurs de `n` et `m`, le calcul ne termine pas en un temps raisonnable voire jamais.

In [ ]:
```
%time chemins(13, 13)
```

Objectif : Utiliser la programmation dynamique pour améliorer l'efficacité de l'algorithme. Vous écrirez une version récursive avec mémoïsation (approche descendante) puis une version itérative avec mémoïsation (approche ascendante).

Version récursive avec mémoïsation (approche descendante)

On va utiliser un tableau `memo` pour mémoriser les résultats des calculs afin de ne pas les effectuer deux fois.

Le tableau `memo` est de taille $(n+1) \times (m+1)$ et l'élément `memo[i][j]` représente le nombre de chemins sur une grille $i \times j$.

On peut créer le tableau `memo` avec uniquement des 1 au départ, puisque c'est la valeur minimale des `memo[i][j]`.

Question 1 : Ecrivez une fonction `chemins_memo(n, m, memo)` qui renvoie le nombre de chemins sur une grille de taille $n \times m$ en utilisant le tableau `memo` pour stocker les résultats calculés afin de ne pas les calculer plusieurs fois.

In [ ]:
```
# à vous de jouer !
```

Question 2 : Ecrivez une fonction `chemins_dyn_desc(n, m)` renvoie le nombre de chemins d'une grille $n \times m$. Il suffit de lancer le premier appel de la fonction `chemins_memo` sur un tableau ne contenant que des 1.

In [ ]:
```
# à vous de jouer !
```

Version itérative (approche ascendante)

On va utiliser un tableau `grille` de taille $(n+1) \times (m+1)$ dans lequel `grille[i][j]` est le nombre de chemins sur une grille de taille $i \times j$.

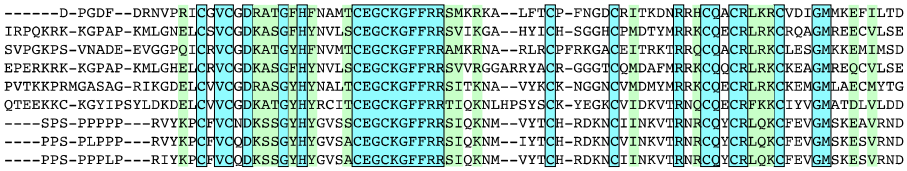
On peut commencer par créer ce tableau `grille` avec uniquement des 1, ce qui permet d'initialiser correctement la première ligne et la première colonne. Ensuite, avec deux boucles `for` imbriquées on peut calculer les valeurs `grille[i][j]` en progressant dans le sens des indices croissants et en utilisant la relation de récurrence.

Question 3 : Ecrivez une fonction `chemin_dyn_asc(n, m)` qui renvoie le nombre de chemins d'une grille de taille $n \times m$ en construisant et complétant le tableau `grille` au fur et à mesure.

In [ ]:
```
# à vous de jouer !
```

Exercice 3 : Alignement de séquences

L'*alignement de séquences* est un problème algorithmique classique de bio-informatique visant typiquement à comparer deux séquences d'ADN dans l'idée que si deux séquences d'ADN représentant des gènes sont similaires, alors la fonction du gène est probablement la même.



Fdardel (https://commons.wikimedia.org/wiki/File:Multalign.svg), CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0), via Wikimedia Commons

Décontextualisation : On peut représenter deux séquences comme deux chaînes de caractères T_1 et T_2 et l'objectif est d'étudier leur degré de similarité (ou dissimilarité). Il y a deux approches possibles :

- soit on étudie leur dissimilarité : on cherche donc à *minimiser* la *distance* $d(T_1, T_2)$ entre les deux chaînes ;
- soit on étudie leur similarité : on cherche donc à *maximiser* un score de ressemblance $s(T_1, T_2)$ entre les deux chaînes.

Dans cet exercice, on choisira la première option : *minimiser* la distance entre deux chaînes.

Distance d'édition et alignement

Pour mesurer la distance entre deux chaînes, on choisit d'utiliser la **distance d'édition**, aussi appelée *distance de Levenshtein* (https://fr.wikipedia.org/wiki/Distance_de_Levenshtein).

L'objectif est de passer d'une chaîne à l'autre au moyen de 3 opérations :

- insertion (d'un caractère c dans T_1)
- suppression (d'un caractère c dans T_1)
- substitution (d'un caractère c par un autre caractère c' dans T_1)

Chacune de ces opérations a un coût et la *distance d'édition*, notée dE, minimise la somme de ces coûts pour passer d'une chaîne à l'autre. Autrement dit, la distance d'édition est égale au nombre minimal de caractères qu'il faut insérer, supprimer, ou substituer pour passer d'une chaîne à l'autre.

Dans la suite, on supposera que les coûts des trois opérations sont tous égaux à 1 !!

Exemple : T_1 = SUCCES et T_2 = ECHECS. On peut considérer l'alignement suivant :

```
S U C C E - S
- E C H E C S
```

Les tirets (-) sur le premier mot correspondent à une supression, et les tirets sur le deuxième mot correspondent à une insertion. Si deux caractères sont alignés : soit il s'agit d'une substitution s'ils sont différents, soit il s'agit d'une correspondance s'ils sont égaux (le coût est nul dans ce cas).

Autrement dit, avec cet alignement, pour passer de SUCCES à ECHECS, on a effectué les opérations suivantes :

- suppression de S : **-**UCCES (coût = 1)
- substitution de U par E : **E**CCES (coût = 1)
- correspondance de C : E**C**CES
- substitution de C par H : EC**H**ES (coût = 1)
- correspondance de E : ECH**E**S
- insertion de C : ECHE**C**S (coût = 1)
- correspondance de S : ECHEC**S**

On obtient un coût total égal à 4 (en fait il s'agit d'un alignement optimal, donc $\mathrm{dE}(\mathrm{SUCCES}, \mathrm{ECHECS}) = 4$, comme on le verra plus tard).

Question 1 : T_1 = SUCCES et T_2 = ECHECS. On considère l'alignement suivant

```
S - U C C E - S
- E - C H E C S
```

1. Ecrivez les opérations pour passer de SUCCES à ECHECS avec cet alignement.
2. Quelle est le coût de cet alignement ? Cet alignement est-il meilleur que le précédent ?

Moralité : Plusieurs alignements sont possibles et parmis ces alignements, certains ont un coût minimal : il s'agit d'alignements optimaux dont le coût est égal à la distance d'édition entre les deux chaîne. Ainsi, trouver un alignement optimal est équivalent à déterminer la distance d'édition.

Relation de récurrence

Le calcul de la distance d'édition entre deux chaînes peut se faire à partir de celles de chaînes plus petites. Cela permet de dégager une relation de récurrence qui sera exploitée par les algorithmes calculant la distance d'édition.

Notations : Dans la suite, on notera:

- n_1 la longueur de T_1 ; n_2 la longueur de T_2
- $T_1[i]$ le i-ème caractère de T_1 ; $T_2[j]$ le j-ème caractère de T_2 (avec $1 \leq i \leq n_1$ et $1 \leq j \leq n_2$)
- $T_1[1..i]$ les i premiers caractères de T_1 ; $T_2[1..j]$ les j premiers caractères de la chaîne T_2, (avec $1 \leq i \leq n_1$ et $1 \leq j \leq n_2$).

Comment peut-on calculer $\mathrm{dE}(T_1[1..i], T_2[1..j])$ si on connait $\mathrm{dE}()$ pour des "bouts de textes" plus courts ?

Supposons avoir aligné $T_1[1..i]$ et $T_2[1..j]$ (les i premiers caractères de T_1 et les j premiers caractères de T_2) et cherchons le coût de cet alignement. On regarde l'alignement le plus à droite. Quatre cas se présentent :

- Suppression : $T_1[i]$ s'aligne sur « - ». Cela demande à avoir aligné $T_1[1..i-1]$ et $T_2[1..j]$ et supprimé $T_1[i]$. On a donc dans ce cas :
$$\mathrm{cout}(T_1[1..i], T_2[1..j]) = \mathrm{cout}(T_1[1..i-1], T_2[1..j]) + \underbrace{1}_{\text{coût suppression}}$$
- Insertion : « - » s'aligne sur $T_2[j]$. Cela demande à avoir aligné $T_1[1..i]$ et $T_2[1..j-1]$ et inséré $T_2[j]$. On a donc dans ce cas :
$$\mathrm{cout}(T_1[1..i], T_2[1..j]) = \mathrm{cout}(T_1[1..i], T_2[1..j-1]) + \underbrace{1}_{\text{coût insertion}}$$
- Substitution : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] \neq T_2[j]$. Cela demande d'avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ et substitué $T_1[i]$ par $T_2[j]$. On a donc dans ce cas :
$$\mathrm{cout}(T_1[1..i], T_2[1..j]) = \mathrm{cout}(T_1[1..i-1], T_2[1..j-1]) + \underbrace{1}_{\text{coût substitution}}$$
- Correspondance : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] = T_2[j]$. Cela demande d'avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ et c'est tout (on substitue le caractère par lui-même). On a donc dans ce cas :
$$\mathrm{cout}(T_1[1..i], T_2[1..j]) = \mathrm{cout}(T_1[1..i-1], T_2[1..j-1]) + \underbrace{0}_{\text{sans surcoût}}$$

Comme la distance d'édition cherche à minimiser le coût d'un alignement, il faut choisir la valeur minimale de coût à chaque fois (parmi les 4 possibilités). On en déduit la relation de récurrence suivante (on a regroupé les deux derniers cas ensemble en introduisant un coût de substitution qui vaut 1 ou 0 selon les cas) :

$$\mathrm{dE}(T_1[1..i], T_2[1..j]) = \min \begin{cases} \mathrm{dE}(T_1[1..i-1], T_2[1..j]) + 1 \\ \mathrm{dE}(T_1[1..i], T_2[1..j-1]) + 1 \\ \mathrm{dE}(T_1[1..i-1], T_2[1..j-1]) + \mathrm{sub}(T_1[i], T_2[j]) \end{cases}$$

où $\mathrm{sub}(a, b) = \begin{cases} 0 \text{ si } a = b \text{ (correspondance)} \\ 1 \text{ si } a \neq b \text{ (substitution)} \end{cases}$

Version récursive naïve

On peut utiliser cette relation de récurrence pour écrire l'algorithme récursif naïf correspondant. Voici une version avec des *slices* qui est plus rapide à écrire (mais dont le coût en espace est plus important).

In [ ]:
```
# Version récursive naïve
def dE(t1, t2):
    if len(t1) == 0:
        return len(t2)
    elif len(t2) == 0:
        return len(t1)
    else:
        if t1[-1] == t2[-1]:
            sub = 0
        else:
            sub = 1
        return min(dE(t1[0:-1], t2) + 1, dE(t1, t2[0:-1]) + 1, dE(t1[0:-1], t2[0:-1]
```

On peut constater que cela marche pour des mots assez courts

In [ ]:
```
%time dE('SUCCES', 'ECHECS')
```

Que cela prend un peu plus de temps si on augmente un peu la longueur des mots.

In [ ]:
```
%time dE('DESTRUCTION', 'CREATION')
```

Et que cela ne termine pas en un temps raisonnable (voire pas du tout) si les mots sont encore un peu plus longs (*Vous pourrez stopper l'exécution manuellement si besoin*).

In [ ]:
```
%time dE('SOUSTRACTION', 'MULTIPLICATION')
```

Question 2 : Dessinez les 3 premiers niveaux de l'arbre d'appels récursifs si on lance `dE('SUCCES', 'ECHECS')` . *On pourra juste noter les deux chaînes en arguments pour gagner un peu de temps.*

> Vous devez constater qu'il y a des appels redondants, et qu'ils deviennent vite beaucoup trop nombreux dès que la longueur des chaînes augmente un peu. L'efficacité est catastrophique !

Programmation dynamique

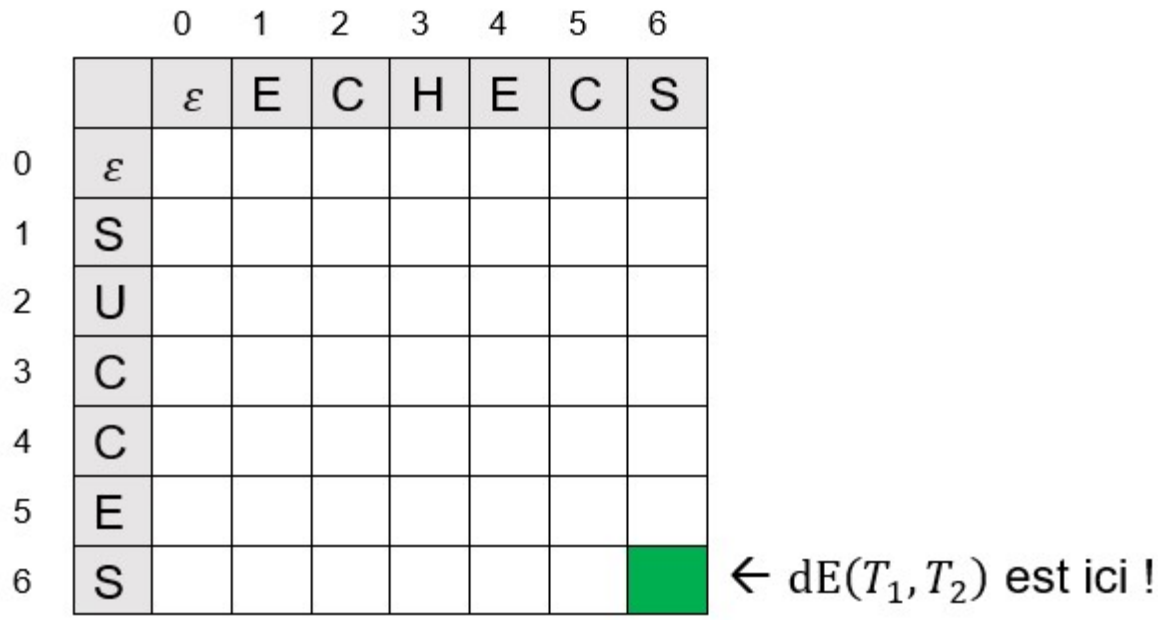
Vous allez maintenant utiliser la programmation dynamique pour améliorer l'algorithme précédents. On ne traitera que la version itérative ascendante (la version récursive avec mémoïsation est laissée en exercice).

Tableau de mémorisation

Comme pour les exemples précédents, l'idée est d'utiliser un tableau pour stocker les réponses déjà connues pour ne pas les calculer plusieurs fois.

Dans le cas de l'alignement de séquence :

- on va utiliser un tableau `D` à deux dimensions de taille $(n1+1) \times (n2+1)$.
- Pour tous $1 \leq i \leq n_1$ et pour tous $1 \leq j \leq n_2$, on a : `D[i][j]` $= \mathrm{dE}(T_1[1..i], T_2[1..j])$ (l'élément en position (i, j) est la distance d'édition entre les i premiers caractères de T_1 et les j premiers caractères T_2)
- La solution $\mathrm{dE}(T_1, T_2)$ se trouve dans la case en bas à droite `D[n1][n2]` .



Remarques :

- On ajouté une première ligne et une première colonne qui permettront de simplifier le début du remplissage du tableau. Cette première ligne contient ε qui désigne un mot vide.
- Dans cet exemple, la case en position $(2, 4)$ c'est-à-dire `D[2][4]` contient la distance d'édition entre les deux premières caractères de `'SUCCES'` et les quatre premiers caractères de `'ECHECS'` , donc elle contient $\mathrm{dE}(\text{'SU'}, \text{'ECHE'})$.

Remplissage du tableau

- On peut remplir facilement la première ligne et la première colonne car on cherche à chaque fois la distance avec un mot vide : il faut insérer ou supprimer le nombre de lettres du second mot. On obtient alors :

		0	1	2	3	4	5	6
		ε	E	C	H	E	C	S
0	ε	0	1	2	3	4	5	6
1	S	1						
2	U	2						
3	C	3						
4	C	4						
5	E	5						
6	S	6						

- Il suffit de remplir ensuite le tableau ligne par ligne et de haut en bas en utilisant la relation de récurrence.

Question 3 : Exprimez `D[i][j]` en fonction de `D[i-1][j]`, `D[i][j-1]` et `D[i-1][j-1]` (avec `i` entre 1 et n_1, et `j` entre 1 et n_2) puis identifiez les 4 cases en question dans le tableau `D`.

Question 4 : Pour compléter une case du tableau on utilise soit celle à sa gauche, soit celle au-dessus, soit celle au-dessus à gauche en diagonale. Selon le cas, indiquez s'il s'agit d'une substitution, d'une suppression, d'une insertion ou d'une correspondance.

Question 5 : Vous devriez avoir tous les éléments pour compléter le tableau `D`. Faites-le et déduisez-en la distance d'édition entre les mots SUCCESS et ECHECS.

Question 6 : Utilisez un tableau pour déterminer la distance d'édition entre les mots NICHE et CHIENS.

Implémentation

Question 7 : Écrivez une fonction `dE(t1, t2)` qui renvoie la distance d'édition entre les chaînes `t1` et `t2` ainsi que le tableau `D` (permettra de vérifier!). Vous utiliserez la méthode itérative ascendante qui construit la table de programmation dynamique `D` comme évoqué dans les question précédentes. *On pourra créer le tableau `D` de départ avec des zéros puis initialiser la première ligne et la première colonne.*

In [ ]:
```
# A vous de jouer !
```

Question 8 : Testez la fonction, en particulier pour vérifier vos réponses aux questions 5 et 6 (distances entre SUCCES et ECHECS, puis entre NICHE et CHIENS. Vous vérifierez également que l'on obtient des réponses rapides mêmes pour des mots plus longs.

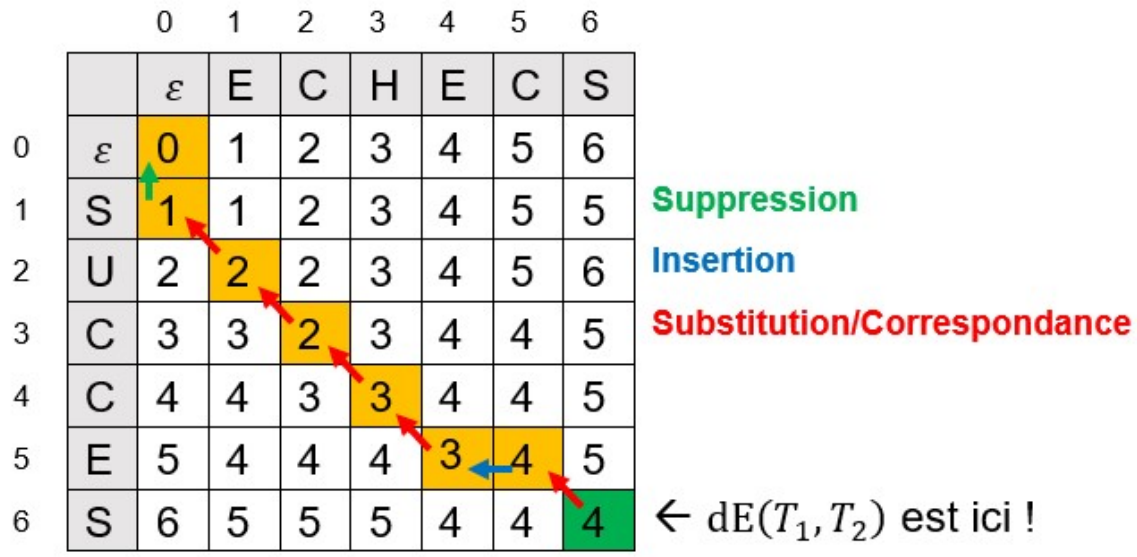
In [ ]:
```
# à vous de jouer !
```

Construction d'un alignement optimal

Pour trouver un alignement optimal entre T_1 et T_2 il suffit de *remonter* depuis `D[n1][n2]` vers `D[0][0]` (de la case en bas à droite à celle en haut à gauche) en considérant à chaque fois le "meilleur" des 3 voisins, c'est-à-dire celui par lequel on a pu arriver ! (il peut parfois y en avoir plusieurs). Il faut se rappeler (cf. question 4) que :

- Si on monte, il s'agit d'une suppression
- Si on va à gauche, il s'agit d'une insertion
- Si on va en diagonale, il s'agit d'une substitution ou d'une correspondance

Par exemple, le tableau `D` correspondant à l'alignement entre SUCCES et ECHECS est :



On peut alors construire un alignement optimal (en partant de la fin ou du début) :

```
S U C C E - S
- E C H E C S
```

Question 9 : Déterminez un autre alignement optimal de ce deux chaînes.

Question 10 : Utilisez le tableau de la question 6 pour trouver un alignement optimal entre NICHE et CHIENS. Y en a-t-il d'autres ?

Question 11 (exercice) : Écrivez une fonction `alignement_optimal(t1, t2)` qiu renvoie un alignement optimal entre les deux chaînes `t1` et `t2` . On peut bien sûr s'appuyer sur la fonction `dE` de la question 7.

Question 12 (exercice) : Écrivez une fonction permettant de calculer la distance d'édition entre deux chaînes en utilisant une version récursive avec mémoïsation.

Références :

- Equipe pédagogique DIU EIL, Université de Nantes.
- Articles Wikipédia sur la Programmation dynamique (https://fr.wikipedia.org/wiki/Programmation_dynamique) et sur la Distance de Levensthein (https://fr.wikipedia.org/wiki/Distance_de_Levenshtein)
- Livre *Spécialité Numérique et sciences informatiques : 24 leçons avec exercices corrigés - Terminale*, éditions Ellipses, T. Balabonski, S. Conchon, J.-C. Filliâtre, K. Nguyen. Site du livre : http://www.nsi-terminale.fr/ (http://www.nsi-terminale.fr/) pour l'idée de l'exercice 2 notamment.
- Livre *Spécialité Numérique et sciences informatiques : 30 leçons avec exercices corrigés - Première*, éditions Ellipses, T. Balabonski, S. Conchon, J.-C. Filliâtre, K. Nguyen. Site du livre : http://www.nsi-premiere.fr/ (http://www.nsi-premiere.fr/) pour la version gloutonne de l'exercice 1.
- Cours de David Roche sur le Pixees : Programmation dynamique (https://pixees.fr/informatiquelycee/n_site/nsi_term_algo_progdyn.html)
- Un cours sur la programmation dynamique : http://www-desir.lip6.fr/~spanjaard/pmwiki/uploads/ProgrammationDynamique.pdf (http://www-desir.lip6.fr/~spanjaard/pmwiki/uploads/ProgrammationDynamique.pdf)

Germain BECKER, Lycée Mounier, ANGERS