T.P. 4 : Solution de la première partie sur MIU

a) Ecrire deux fonctions `R1` et `R2` qui prennent une chaîne de caractères `S` en argument et retournent la chaîne obtenue par l'application des règles `R1` et `R2` à `S`.

```
def R1(S):
    if S[-1]=="I": # S[-1] permet d'accéder à la dernière entrée, c'est très pratique.
        return S+"U"
def R2(S):
    return S+S[1:]
```

Remarques : • Noter que si `S` ne se termine pas par un `I`, la fonction `R1` ne retournera rien, c'est-à-dire retournera un `None`.
Tous les mots qu'on manipule ici commencent par un `M`. Pour `R2` on supposera déjà que le mot passé en argument commence par `M`.

b) Ecrire une fonction `R3(S,i)` qui prend en argument une chaîne de caractères `S` et un indice `i` et qui regarde si la chaine `S` contient la chaîne `III` aux entrées `S[i], S[i+1], S[i+2]` et si c'est le cas, retourne la chaîne de caractères déduite de `S` en remplaçant ces `III` suivant la règle 3.
N.B. Contrairement aux listes, les chaînes de caractères ne sont pas modifiables, donc on va créer une autre chaîne.

```
def R3(S,i):
    if i<=len(S)-3 and S[i:i+3]=="III":
        T=S[0:i]+"U"+S[i+3:]
        return T
```

Remarque : bien noter l'ordre des conditions dans le `if i<=len(S)-3 and S[i:i+3]=="III":` qui tient compte du *caractère paresseux du* `and`.
Si on avait mis `if S[i:i+3]=="III" and i<=len(S)-3:` on aurait eu un `out of range` pour `i>len(S)-3`.

c) Ecrire une fonction `appliqueR3` qui prend en argument une chaine de caractères `S` et qui renvoie la liste de tous les mots qu'on peut obtenir en appliquant *une seule fois* `R3` à `S`.

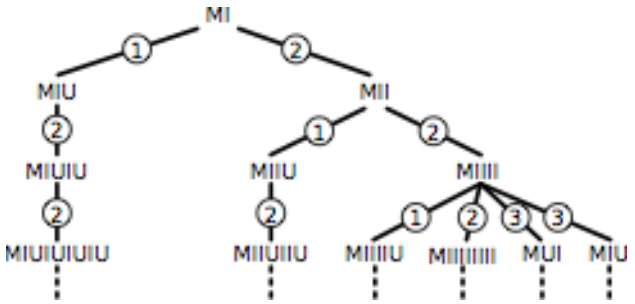
```
def appliqueR3(S):
    L=[] # on va fabriquer une liste
    if S!=None: # cette condition, superflue à ce stade, sera utile pour MIU
        for i in range(len(S)-2):
            if R3(S,i)!=None:
                L.append(R3(S,i))
    return L
```

d) Faire de même une fonction `R4(S,i)` et une fonction `appliqueR4(S)`.

```
def R4(S,i):
    if S!=None:
        if i<=len(S)-2 and S[i:i+2]=="UU":
            return S[0:i]+S[i+2:]


def appliqueR4(S):
    L=[]
    for i in range(len(S)-1):
        if R4(S,i)!=None:
            L.append(R4(S,i))
    return L
```

e) Ecrire enfin une fonction `MIU(n)` qui prend en argument un entier `n` et renvoie la liste de tous les mots qu'on peut obtenir à partir de `MI` en appliquant `n` fois les règles du système comme indiqué dans l'arbre suivant :



Ainsi par exemple :

```
>>> MIU(1)
['MI', 'MIU', 'MII']

>>> MIU(2)
['MI', 'MIU', 'MII', 'MIUIU', 'MIIU', 'MIIII']
```

Comme expliqué, en TP, il est important, pour obtenir la n-ième génération de mots, de n'appliquer les règles de constructions qu'aux mots de la $n-1$-ième génération. Cela nous incite à utiliser trois listes (ou bien une seule liste de listes, mais par simplicité à notre niveau ici je préférerais trois listes).

Dans le code qui suit `L` contiendra la liste totale. Ensuite on a deux listes `Lcourant` et `Lnouvelle`. A chaque étape, on parcourt `Lcourant` (qui correspond à la $n-1$-ième génération), on applique les quatre règles à chaque mot de `Lcourant` et on stocke le résultat dans `Lnouvelle` (la n-ième génération). Pour itérer, `Lnouvelle` devient `Lcourant`.

```
def MIU(n):
    #n est le nombre d'étapes d'application
    L=["MI"]
    Lcourant=L
    for i in range(n):
        Lnouvelle=[]
        for chaine in Lcourant:
            if R1(chaine) not in L and R1(chaine)!=None:
                Lnouvelle.append(R1(chaine))
            if R2(chaine) not in L:
                Lnouvelle.append(R2(chaine))
            Lnouvelle=Lnouvelle+appliqueR3(chaine)
            Lnouvelle=Lnouvelle+appliqueR4(chaine)
        L=L+Lnouvelle
        Lcourant=Lnouvelle
    return L
```