Correction du TD itc² n° 8: Programmation dynamique: Plus longue sous-séquence commune par diverses approches

1 Approche par force brute

1. On propose la fonction suivante qui initialise un compteur ik à 0, puis itère sur la chaine de caractères Y par une boucle `for` en vérifiant pour chaque caractère s'il est présent dans la chaine Yk; si c'est le cas, on incrémente le compteur ik de position dans Yk, sinon le compteur est inchangé. En fin d'itération, si Yk est une sous-séquence de Y alors le compteur ik doit être égal à la longueur de la chaine Yk; dans ces conditions, le code renvoie `True`, et sinon `False`.

Listing 1:

```python
def verif(Yk,Y):
    ik=0
    for car in Y:
        if ik<len(Yk) and Yk[ik]==car:
            ik+=1
    if ik==len(Yk):
        return True
    return False
```

2. Pour chaque élément de X, on peut choisir de l'intégrer ou non dans une sous-séquence; par conséquent, si X comporte n caractères, alors le nombre de sous-séquences possibles est 2^n; un code générant ces sous-séquences auraient donc une complexité $\mathcal{O}(2^n)$. Pour chacune de ces sous-séquences, la vérification qu'elle est une sous-séquence commune à Y se ferait en $\mathcal{O}(m)$ d'après ce qui précède, il faudrait encore prendre la plus longue sous-séquence commune; finalement un algorithme de force brute serait de complexité $C(n,m) = \mathcal{O}(m \times 2^n)$.

2 Approche récursive

3. Pour $i = 0$ ou $j = 0$, il n'y a pas de sous-séquence commune donc $L(0, j) = L(i, 0) = 0$. Pour $(i, j) > (0, 0)$, on a:

   - si $x_i = y_j$ alors la taille de la sous séquence commune augmente d'une unité donc: $L(i, j) = 1 + L(i-1, j-1)$
   - si $x_i \neq y_j$ alors la taille de la sous séquence commune est $max(L(i-1, j), L(i, j-1))$

4. On propose la fonction récursive suivante:

Listing 2:

```python
def Long_PLSSC_rec(X,Y):
    if len(X)==0 or len(Y)==0: #cas de base
        return 0 # sous-séquence commune de longueur nulle
    elif X[-1]==Y[-1]: # 1er cas de la récurrence ie égalité des derniers caractères de X et de Y
        return 1+Long_PLSSC_rec(X[:-1],Y[:-1]) # ajoute 1 à la longueur de la sous-séquence commune précédente que l'on calcule récursivement
    else:
        return max(Long_PLSSC_rec(X[:-1],Y), Long_PLSSC_rec(X,Y[:-1])) # sinon 2nd cas de la récurrence
```

5. La complexité (en terme de nombre d'appels récursifs) se calcule sans difficulté. Dans le pire des cas, pour lequel les deux séquences n'ont aucun caractère en commun, c'est systématiquement le second cas de récursion (récursion double) qui s'applique, or celui-ci induit deux appels récursifs dans lesquels l'une des deux sous-séquences est réduite d'un caractère; le cas de base sera atteint dans le pire des cas, lorsque la plus longue de deux chaînes, X ou Y, sera "épuisée" par slicing on a donc par exemple en supposant $n < m$:

$$C(n) = 2C(m-1) = 2^2C(m-2) = 2^mC(0)$$

3 Approche par programmation dynamique

6. La récurrence dégagée plus haut montre que la recherche de la longueur de **la plus longue** sous-séquence commune $L(i, j)$ (entre X_i et Y_j) passe nécessairement par la détermination des longueurs des plus longues sous-séquences communes $L(i-1, j-1)$, $L(i-1, j)$, et $L(i, j-1)$. Le problème présente donc une sous-structure optimale.

7. Le remplissage du tableau donne:

	long. séquence Y →	0	1	2	3	4	5	6
long. séquence X ↓	séquence X↓ / séquence Y→		B	A	C	B	D	A
0			**0**	**0**	**0**	**0**	**0**	**0**
1	A	**0**	0	1	1	1	1	1
2	B	**0**	1	1	1	2	2	2
3	C	**0**	1	1	2	2	2	2
4	B	**0**	1	1	2	3	3	3
5	A	**0**	1	2	2	3	3	4

La valeur de la plus longue sous-séquence commune se lit en $L[n, m]$, donc ici $L[n, m] = 4$.

8. On propose la fonction suivante qui exploite la récurrence dégagée plus haut:

Listing 3:

```python
def Long_PLSSC_dyn(X,Y):
    n=len(X)
    m=len(Y)
    L=np.zeros((n+1,m+1),dtype=int)
    for i in range(1,n+1): #on parcourt la chaine X
        for j in range(1,m+1): #on parcourt la chaine Y
            if X[i-1]==Y[j-1]: #si les derniers caractères sont identiques
                L[i,j]=1+L[i-1,j-1] # alors on incrémente la longueur
            else:
                L[i,j]=max(L[i-1,j],L[i,j-1]) # sinon on prend la valeur de longueur maximale entre ces deux possibilités
    return L[n,m]
```

9. Aucune difficulté sur cette question: les deux boucles imbriquées conduisent à une complexité en $\mathcal{O}(n \times m)$.

10. <u>Principe de lecture du tableau L:</u>

On part de $L[n, m]$, et on remonte dans le tableau en exploitant encore la récurrence dégagée plus haut, avec:

- Si $L[n, m] = L[n-1, m]$ alors $X[n-1]$ ne fait pas partie de la PLSSC, et on se place en $L[n-1, m]$.
- Si $L[n, m] = L[n, m-1]$ alors $Y[m-1]$ ne fait pas partie de la PLSSC, et on se place en $L[n, m-1]$.
- Sinon, il faut intégrer $X[n-1] = Y[m-1]$ à la PLSSC, et on se place en $L[n-1, m-1]$
- Lorsque $n = 0$ ou $m = 0$, alors le parcours de l'une des deux chaines est achevé et on s'arrête!

11. On propose la fonction suivante qui suit la démarche ci-dessus:

Listing 4:

```python
def PLSCC(X,Y):
    L=Long_PLSSC_dyn(X,Y)
    n,m=len(X),len(Y)
    PLSCC=""
    while n>0 and m>0:
        if L[n,m]==L[n-1,m]:
            n-=1
        elif L[n,m]==L[n,m-1]:
            m-=1
        else:
            n-=1
            m-=1
            PLSCC=X[n]+PLSCC
    return PLSCC
```