Correction du problème de recherche de plus longue sous-séquence commune

Exercice 1

substr = ACTAGACTAG

Force brute

3. a. i est un indice indiquant la position dans la chaine substr et j est un indice indiquant la position dans la chaine str2.

b. On recherche la présence du caractère c dans la chaine str2. A la sortie de la boucle while précédent cette ligne, ou bien on a trouvé le caractère c, auquel cas on passe à la recherche du caractère suivant, ou bien j est égal à n2, ce qui signifie qu'on a parcouru toutes les lettres de str2 sans avoir trouvé le caractère c. Dans ce cas, la sous-chaine substr n'est pas extraite de str2 et on renvoie False.

4. On teste 2^{n1} sous-chaines de str1 (sauf la chaine vide éventuellement). La constitution d'une sous-chaine est en O(n1). Le test d'appartenance à str2 est O(len(substr) + n2).

Comme len(substr) ≤ n1, on peut dire que le test d'appartenance est en O(n1 + n2).

La complexité de cet algorithme est en $O(2^{n1}(n1 + n2))$.

Deux commentaires :

- On a intérêt à choisir pour str1 la plus petite des deux chaines ou se contenter de tester les sous-chaines de str1 de longueur inférieure à la longueur de str2.
- Cet algorithme a une complexité exponentielle et devient vite inapplicable.
  Si n1 = 60, $2^{60} \approx 10^{18}$, ce qui prend de l'ordre de 30 ans.

6. On pourrait améliorer cet algorithme en commençant par tester les sous-chaines de longueurs maximales et en interrompant l'exécution dès qu'on trouve une sous-chaine qui convient, mais il vaut mieux se concentrer sur la programmation dynamique.

Exercice 2

Compléter le tableau correspondant à la recherche d'alignement de séquence par programmation dynamique avec str1 = GCAGTCA et str2 = CACGCTA.

str1\str2		C	CA	CAC	CACG	CACGC	CACGCT	CACGCTA
G					G	G	G	G
GC		C	C	C	C/G	GC	GC	GC
GCA		C	CA	CA	CA	CA/GC	CA/GC	GCA
GCAG		C	CA	CA	CAG	CAG	CAG	CAG/GCA
GCAGT		C	CA	CA	CAG	CAG	CAGT	CAGT
GCAGTC		C	CA	CAC	CAC/CAG	CAGC	CAGC/CAGT	CAGC/CAGT
GCAGTCA		C	CA	CAC	CAC/CAG	CAGC	CAGC/CAGT	CAGCA/CAGTA

Algorithme

1.

```
pour j allant de 0 à n2 :
        T[0, j] ← ""

pour i allant de 1 à n1 :
        T[i, 0] ← ""

pour i allant de 1 à n1 :
        pour j allant de 1 à n2 :
                si str1[i-1] = str2[j-1] alors :
                        T[i, j] ← T[i-1, j-1] + str1[i-1]
                sinon :
                        T[i, j] ← maxi(T[i, j-1], T[i-1, j])        # maxi(str1, str2) renvoie la chaine la plus longue
```

Suite à l'exécution de cet algorithme, T[n1, n2] contient une chaine de longueur maximale.

2. On a deux boucles imbriquées, de longueurs n1 et n2, donc une complexité en $O(n1 \times n2 \times c)$ où c est la complexité des opérations internes aux deux boucles, des recopies de chaines de caractères. Or, ces chaines de caractères sont au plus de longueur max(n1, n2).
La complexité de cet algorithme est donc en $O(n1 \times n2 \times \max(n1, n2))$.
Si on suppose que les deux chaines de caractères sont de même longueur n, la complexité est en $O(n^3)$.

La complexité en mémoire est du même ordre, mais on peut remarquer qu'on construit une ligne à partir de la ligne précédente uniquement. On pourrait donc modifier notre algorithme pour mémoriser uniquement deux lignes, voire même une seule ligne en conservant en plus le coefficient T[i-1, j-1] si on réécrit la ligne i+1 par-dessus la ligne i.
La complexité en mémoire n'est alors plus qu'en $O(n1 \times \max(n1, n2))$.

Compléments

Comme avec le problème du sac à dos et celui du rendu de monnaie, il est possible d'enregistrer dans le tableau ou le dictionnaire seulement la longueur de la plus grande sous-chaine commune, et de reconstituer ensuite la solution. Dans ce cas, bien sûr, on ne peut pas utiliser le tableau à une seule ligne.