EPU 4 – BIMB *Algorithmique pour la biologie*
Examen du 20 mai 2019 Documents autorisés.

Inscrivez **lisiblement** vos NOM et Prénom en tête de vos copies.

Exercice 1 : (Programmation dynamique : Plus longue sous-séquence croissante [11 points])

Étant donné une séquence de n nombres différents, on souhaite y trouver l'une des plus longues sous-séquences croissantes. Par exemple si la séquence fournie est `[10, 53, 85, 35, 51, 52, 74, 52, 07, 20, 71, 75, 13]`, alors la plus longue sous-séquence croissante est `[10, 35, 51, 52, 52, 71, 75]`. (Notez que la croissante n'est pas forcément stricte). Si plusieurs sous-séquences possèdent la taille la plus longue, l'algorithme peut retourner n'importe laquelle.

On note S la séquence d'entrée complète (de n éléments numérotés de 0 à $n-1$), $S[i]$ son $(i+1)^{\text{ème}}$ élément et $S[0..i]$ son $(i+1)^{\text{ème}}$ préfixe (c'est-à-dire la séquence constituée seulement des $i+1$ premiers éléments).

1. **[2 points]** Une première idée consiste à se ramener à l'algorithme de recherche de la plus longue sous-séquence commune à deux séquences : pour deux chaînes u et v de tailles $|u|$ et $|v|$, l'idée est d'utiliser l'algorithme de Needleman et Wunsch avec une pénalité linéaire nulle ($g = 0$) et une matrice de substitution égale à la matrice identité.
   La recherche de la plus longue sous-séquence croissante peut alors s'écrire ainsi :

   ```
   def PlusLongueSousSéquenceCroissante(S):
       return (PlusLongueSousSéquenceCommune(S, TriCroissant(S)))
   ```

   où `TriCroissant` permet de trier une liste par ordre croissant. Donnez le nom d'un algorithme de tri qui pourrait être utilisé ici et indiquez la complexité de cet algorithme de tri. Donnez ensuite la complexité de `PlusLongueSousSéquenceCroissante` pour l'algorithme de tri que vous avez choisi. (Vous donnerez ces complexités en fonction de $n = |S|$)

On souhaite maintenant développer un algorithme de programmation dynamique qui calcule la plus longue sous-séquence croissante sans avoir besoin ni d'effectuer un tri, ni d'appeler `PlusLongueSousSéquenceCommune`.

Pour $k < n$, notons $L[k]$ la longueur de la plus longue sous-séquence croissante de $S[0..k]$ qui se termine par $S[k]$. On a bien sûr $L[0] = 1$. Selon l'ordre de $S[0]$ et $S[1]$, la valeur de $L[1]$ sera soit $L[0] + 1$ (lorsque $S[0] < S[1]$), soit 1 (lorsque $S[1] < S[0]$).

2. **[2 points]** Donnez la relation de récurrence qui exprime $L[k]$ en fonction des termes précédents de L, et des éléments de S.
3. **[1 point]** Une fois que les $L[0], L[1], ..., L[n-1]$ ont été calculés, comment peut-on trouver la taille de la plus longue sous-séquence croissante de S ? (Cette plus longue sous-séquence ne se termine pas forcément par $S[n-1]$.)
4. **[1 point]** Quelle est la complexité de calculer la taille de la plus longue sous-séquence croissante d'une séquence de n éléments, avec un algorithme de programmation dynamique basé sur les définitions précédentes ?
5. **[3 points]** Ecrivez une fonction python qui implémente cet algorithme de programmation dynamique.
6. **[2 points]** Comment faudrait-il adapter cet algorithme pour retourner, en plus de la longueur de la plus longue sous-séquence croissante, la sous-séquence croissante elle-même ? (On ne vous demande pas d'écrire l'algorithme, mais seulement d'expliquer ce qu'il doit faire en plus.)

Exercice 2 : (Plus longue sous-séquence croissante : algorithme itératif [10 points])

Pour développer un algorithme plus direct pour calculer la plus longue sous-séquence croissante, l'idée est de maintenir un ensemble de suites candidates à pouvoir devenir la plus longue. On va ne retenir qu'une seule séquence candidate par longueur : celles qui finissent par l'élément le plus petit. A la lecture du premier nombre $S[0]$, il n'y a qu'une liste candidate de longueur 1 : $[S[0]]$.
Lorsqu'on rajoute un nombre $S[1]$, si $S[1] > S[0]$, alors on étend la liste précedente et on a une liste candidate de longueur 1 ($[S[0]]$) et une liste de longueur 2 ($[S[0], S[1]]$). Si $S[1] < S[0]$, on ne retient qu'une seule liste de longueur 1 ($[S[1]]$).

L'algorithme se base sur la stratégie suivante :

i) on inilisalise la liste des listes candidates à `[[],[]...[]]` dont la longueur est égale au nombres d'éléments de la séquence passée en argument.

ii) si le nouveau nombre $S[i]$ est plus petit que tous les derniers nombres des listes *candidates*, il faut commencer une nouvelle liste candidate de longueur 1 ne contenant que ce dernier nombre. Elle écrase donc la précedente liste candidate de longueur 1.

iii) si le nouveau nombre $S[i]$ est plus grand que tous les derniers nombres de toutes les listes *candidates*, il peut être rajouter à toutes les listes candidates. Et donc, il faut copier la plus longue liste *candidate* (de longueur l), et lui rajouter ce nouveau nombre. Cette nouvelle liste va donner une nouvelle liste candidate de longueur égale à $l+1$.

iv) Si le nouveau nombre $S[i]$ est entre ces deux extrèmes (cas *ii* et *iii*), on recherche le plus grand nombre parmi tous les derniers nombres des listes candidates qui reste cependant inférieur à $S[i]$. Soit l la longueur de cette liste. On copie cette liste candidate et on la rallonge avec l'élément $S[i]$. Cette liste vient écraser la précedente liste candidate de longueur $l+1$.

Notons que la condition suivante est préservée tout au long du processus de création des listes candidates : *le dernier élément de la liste de longeur l, est plus petit que les derniers éléments des listes de longueurs strictement supérieures à l.*

Considérons l'exemple suivant : $S = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3]$.L'éxécution de l'algorithme donne les listes de listes candidates suivantes (par soucis de concision, nous ne représentons que les listes candidates non vides) :

$S[0] = 0$	cas *ii*	[[0]]
$S[1] = 8$	cas *iii*	[[0], [0, 8]]
$S[2] = 4$	cas *iv*	[[0], [0, 4]]
$S[3] = 12$	cas *iii*	[[0], [0, 4], [0, 4, 12]]
$S[4] = 2$	cas *iv*	[[0], [0, 2], [0, 4, 12]]
$S[5] = 10$	cas *iv*	[[0], [0, 2], [0, 2, 10]]
$S[6] = 6$	cas *iv*	[[0], [0, 2], [0, 2, 6]]
$S[7] = 14$	cas *iii*	[[0], [0, 2], [0, 2, 6], [0, 2, 6, 14]]
$S[8] = 1$	cas *iv*	[[0], [0, 1], [0, 2, 6], [0, 2, 6, 14]]
$S[9] = 9$	cas *iv*	[[0], [0, 1], [0, 2, 6], [0, 2, 6, 9]]
$S[10] = 5$	cas *iv*	[[0], [0, 1], [0, 1, 5], [0, 2, 6, 9]]
$S[11] = 13$	cas *iii*	[[0], [0, 1], [0, 1, 5], [0, 2, 6, 9], [0, 2, 6, 9, 13]]
$S[12] = 3$	cas *iv*	[[0], [0, 1], [0, 1, 3], [0, 2, 6, 9], [0, 2, 6, 9, 13]]

1. **[3 points]** Ecrivez une fonction `SearchIndex(Loflists, l, r, elt)` qui prend en argument : une liste de listes candidates, deux index `l` et `r`, et un élément `elt`, et qui renvoie l'indice $i \in [l, r]$ de la liste candidate dont le dernier élément est le plus grand des derniers éléments qui sont inférieurs à `elt`. On pourra présenter au choix une implémentation itérative ou récursive.
   Remarque : à tout moment, si on extrait de chaque liste candidate son dernier élément, et si on met ces derniers éléments dans une liste, on obtient une liste triée.

2. **[5 points]** Implémentez l'algorithme présenté. Après l'initialisation de la liste des listes candidates, on passera en revue chacun des nombres de la liste de nombres passée en paramètres, et pour chacun de ces nombres, on testera pour savoir s'il faut utiliser le cas *ii)*, *iii)* ou *iv)*.
   Pour faciliter le traitement des cas *ii)* et *iii)*, on pourra stocker dans 3 variables différentes :
   — l'indice de la plus longue liste candidate,
   — la valeur du dernier élément le plus petit,
   — la valeur du dernier élément le plus grand.

3. **[2 points]** La figure ci-contre montre la courbe du temps d'exécution de l'algorithme de l'exercice 1 (rouge) et de l'algorithme développé dans cet exercice (bleu). L'algorithme itératif semble bien meilleur. Interprétez ces deux courbes en discutant la complexité de l'algorithme itératif. Justifiez votre réponse. (On considérera que la copie des listes candidates se fait en temps constant.)