UE Ouverture **Devoir de Programmation**

---

*Devoir à faire en **binôme**. Langage de programmation **OCaml**.*
Les enseignants décideront d'un ordre de passage pour une soutenance le mercredi 16/11/2022.
Rapport et Code Source à rendre dans une archive nommée Nom1_Nom2_OUV.zip *(sous moodle) au plus tard le dimanche 13/11/2022 à 23h59.*

Détection de plagiat flagrant

1 Présentation

Le but du problème consiste à résoudre un problème algorithmique d'exploration de chaînes de caractères en utilisant des structures de données plus ou moins évoluées et en comparant les performances de chacune des solutions proposées.
Il est attendu un soin particulier concernant la programmation, dans le paradigme fonctionnel, et par rapport à la réflexion et la mise en place des expérimentations.

Voilà l'énoncé du problème, il s'agit d'une version simplifiée de détection de plagiat :
Étant donné deux chaînes de caractères, notées C_1 et C_2, le problème consiste à chercher les sous-chaînes communes à C_1 et C_2 étant de plus grande longueur. De plus il faudra indiquer leurs positions dans C_1 et C_2.

Rappelons qu'une sous-chaîne d'une chaîne de caractères C est constituée de caractères consécutifs dans C.

`Par exemple, étant donné les deux chaînes ANANAS et BANANE, il existe une seule sous-chaîne commune la plus longue, il s'agit de ANAN.`

La méthode que nous allons mettre en œuvre dans ce devoir consiste à chercher l'un des plus longs préfixes communs pour chaque paire de suffixes de C_1 et C_2.

2 Programmation dynamique

Nous allons commencer par approcher le problème à travers la programmation dynamique. Dans un premier temps nous ne considérons que la longueur des solutions mais pas leur emplacement dans les chaînes C_1 et C_2.

Question 2.1 Déterminer une sous-structure optimale permettant de calculer la longueur des plus longs préfixes communs des suffixes de C_1 et C_2.

Question 2.2 Comment calculer la longueur des sous-chaînes les plus longues communes à C_1 et C_2, en fonction de la sous-structure optimale précédente ?

Question 2.3 Caractériser (par une équation) cette sous-structure optimale.

Question 2.4 Enrichir la solution précédente afin de pouvoir retrouver les positions de d'une sous-chaîne parmi les plus longues dans les deux chaînes C_1 et C_2.

Question 2.5 Proposer une analyse théorique de complexité (au pire cas) de la méthode.
Rappel : avant de calculer la complexité, il faut indiquer ce que l'on mesure pour obtenir la complexité de l'algorithme.

Question 2.6 Implémenter la méthode par programmation dynamique et proposer un jeu de tests.

3 Arbre des suffixes

Dans cette partie nous allons définir, pour une chaîne C, une structure arborescente qui stocke tous les suffixes de cette chaîne. On commence par marquer la fin de la chaîne avec un caractère particulier qui n'apparait pas ailleurs dans la chaîne, par exemple #.

`Par exemple, étant donné la chaîne ANANAS#, la structure arborescente va contenir les chaînes ANANAS#, NANAS#, ANAS#, NAS#, AS#, S# et #.`

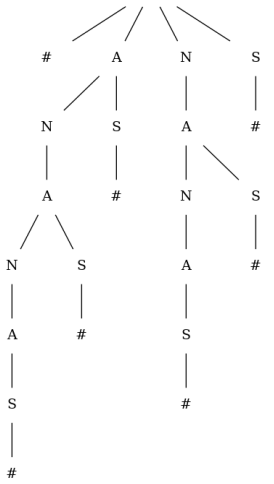

FIGURE 1 – L'arbre des suffixes de ANANAS#

Question 3.7 Étant donné deux chaînes de caractères C_1 et C_2. Comment savoir si C_2 est une sous-chaîne de C_1, en utilisant l'arbre des suffixes de C_1 ?

Question 3.8 Définir une structure de données et les primitives permettant de manipuler les arbres des suffixes.

Question 3.9 Définir une fonction `ArbreSuffixes` prenant en entrée une chaîne de caractères (se terminant par #) et construisant l'arbre des suffixes de cette chaîne.

Question 3.10 Définir une fonction `SousChaine` prenant en entrée deux chaînes de caractères (la seconde ne se termine pas forcément par #) et testant si la seconde est une sous-chaine de la première.

Question 3.11 Proposer un jeu de tests des deux fonctions précédentes.

Question 3.12 Proposer une analyse théorique de complexité (au pire cas) des deux algorithmes implémentés ci-dessus. Il faudra considérer séparément la complexité temporelle et la complexité en espace.

Question 3.13 Définir une fonction `SousChainesCommunes` prenant en entrée deux chaînes, construisant un seul arbre de tous leurs suffixes (et ajoutant un petit peu d'information en chaque nœud) qui détermine la longueur des plus longues sous-chaînes communes et qui renvoie une de ces plus longues sous-chaines.

Question 3.14 Proposer une analyse théorique de complexité (au pire cas), en temps et en espace.

4 Compression de l'arbre des suffixes

Afin de compresser l'arbre des suffixes et ainsi d'améliorer la complexité prouvée en Question 3.14) si un nœud parent n'a qu'un seul enfant, alors on fusionne les deux nœuds (le parent et l'enfant), en un seul dont la clé est la concaténation des caractères. On compresse ainsi l'arbre des suffixes jusqu'à ce que le résultat n'ait plus aucun nœud d'arité 1.

En compressant l'exemple présenté en Figure 1 on obtient la Figure 2.

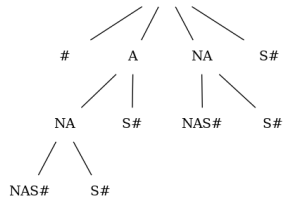

FIGURE 2 – L'arbre des suffixes compacté de ANANAS#

Question 4.15 Implémenter une fonction `compression` prenant en entrée un arbre des suffixes et le compressant avec la règle précédente.

Question 4.16 Pour quelle raison a-t-on ajouté le symbole # à la chaîne pour laquelle nous construisons l'arbre des suffixes compressé ?

Question 4.17 Adapter la fonction `SousChainesCommunes` pour utiliser une structure compressée.

Question 4.18 Étant donné une chaîne C (se terminant par #) de longueur n, quel est le nombre maximal de nœuds de l'arbre des suffixes compressé ? (justifier la réponse)

Question 4.19 Le nombre de nœuds de l'arbre n'est pas l'unique point auquel il faut faire attention dans cette structure de données. En effet, en chaque nœud, on stocke une chaîne de caractère pouvant être de longueur proportionnelle à n. Afin de remédier à cela, plutôt que de stocker la sous-chaîne dans un nœud, on stockera un couple d'entiers correspondant à son indice de départ et de fin dans la chaîne globale. Par exemple le contenu de la feuille en bas à gauche de la Figure 2 contenant $NAS\#$ sera remplacé par $(3, 6)$.
Attention, lorsqu'on parcourt une branche dans l'arbre, on continue de tester l'égalité de caractères, et pas l'égalité de couples d'indices. En effet, les deux occurrences de NA dans la Figure 2 peuvent être associée au même couple, mais pas nécessairement, ce pourrait être $(1, 2)$ *ou* $(3, 4)$.

Question 4.20 Proposer une analyse théorique de complexité (au pire cas), en temps et en espace de la nouvelle fonction de recherche de sous-chaînes.

5 Construction efficace de l'arbre des suffixes compressé

En insérant les suffixes du plus long au moins long dans la structure compressée, on peut construire l'arbre compressé directement.

Question 5.21 Définir une fonction `ArbreSuffixesCompresse` prenant en entrée une chaîne de caractères (se terminant par #) et construisant directement l'arbre des suffixes compressé.

Question 5.22 Proposer un jeu de tests pour cette fonction.

Question 5.23 Proposer une analyse théorique de complexité (au pire cas) , en temps et en espace, de l'algorithme implémenté ci-dessus.

6 Expérimentations

Sur une page du site web de l'Université McGill, différentes réponses à une étude de texte sont proposées et analysées suivant leur degré de plagiat. Le fichier `plagiat.zip` est une archive des exemples présentés. Le fichier `donnee0.txt` contient le texte originel et les autres fichiers contiennent les exemples de réponse. Le but de cette section consiste à comparer les longueur des sous-chaines communes du texte originel par rapport aux différentes réponses. On mettra en regard nos résultats vis-à-vis des explications proposées sur la page web de McGill.

Question 6.24 Extraire les 150 premiers caractères de `donnee0.txt` et de `donnee1.txt` et utiliser les 2 approches par programmation dynamique et via l'arbre des suffixes compressé pour exhiber la plus longue sous-chaîne commune : quelle est elle ?

Question 6.25 Itérer le processus précédent (sur les mêmes fichiers) pour des préfixes de longueurs $[10, 20, 50, 75, 100, 125, 150, 200, 350, 400, 500,$ taille totale$]$ (si ça prend plus d'une minute pour une longueur donnée, arrêter l'expérimentation à cet endroit ; Représenter le temps de calcul nécessaire sur un graphique. Retrouve-t-on la courbe théorique issue de l'analyse de complexité ?

Question 6.26 Répéter la question précédente sur `donnee0.txt` et chacun des autres fichiers.

Question 6.27 (Facultatif) Le but de cette question consiste à obtenir une visualisation des structures de données précédentes. Étant donné un arbre des suffixes (compressé ou non), écrire une fonction `dot` qui construit un fichier représentant le graphe en langage *dot*. Une fois un fichier *dot* construit, l'application *graphviz* permet d'en donner la visualisation.
Des exemples de représentation en *dot* sont présentés ici : `https://graphs.grevian.org/example`.