




Segmentation d'un texte

Préambule

Ce sujet d'oral d'informatique est à traiter, sauf mention contraire, en respectant l'ordre du document. Votre examinatrice ou votre examinateur peut vous proposer en cours d'épreuve de traiter une autre partie, afin d'évaluer au mieux vos compétences.

Le sujet comporte plusieurs types de questions. Les questions sont différenciées par une icône au début de leur intitulé :

- les questions marquées avec  nécessitent d'écrire un programme dans le langage demandé. Le jury sera attentif à la clarté du style de programmation, à la qualité du code produit et au fait qu'il compile et s'exécute correctement ;
- les questions marquées avec  sont des questions à préparer pour présenter la réponse à l'oral lors d'un passage de l'examinatrice ou l'examinateur. Sauf indication contraire, elles ne nécessitent pas d'appeler immédiatement l'examinatrice ou l'examinateur. Une fois la réponse préparée, vous pouvez aborder les questions suivantes ;
- les questions marquées avec  sont à rédiger sur une feuille, qui sera remise au jury en fin d'épreuve.

Votre examinatrice ou votre examinateur effectuera au cours de l'épreuve des passages fréquents pour suivre votre avancement. En cas de besoin, vous pouvez signaler que vous sollicitez explicitement son passage. Cette demande sera satisfaite en tenant également compte des contraintes d'évaluation des autres candidates et candidats.

I Introduction

Dans ce sujet, on s'intéresse à la *segmentation* d'un texte en *mots*.

I.A – Exemple introductif

Imaginons que l'on dispose d'un texte en français dont les espaces qui séparent habituellement les mots ont été oubliés. La **figure I.1** présente le chapeau de la préface du roman *Les Trois Mousquetaires* d'Alexandre DUMAS dans lequel les espaces ont été omis.

```
préfacédanslaquelleilestétablique,malgréleursnomsen  
osetenis,leshérosdel'histoirequenousallonsavoirl'ho  
mneurderaconteranoslecteursn'ontriendemythologique.
```

Figure I.1 Chapeau de la préface de *Les trois mousquetaires* en *scripto continua*

Est-il possible, à partir de ce texte, de retrouver les mots qui le composent initialement ?

I.B – Contexte

Si l'exemple précédent peut sembler quelque peu artificiel — on ne voit pas trop pourquoi on perdrait les espaces d'un texte en français, ce problème se pose réellement pour certaines langues qui n'utilisent pas toujours de symboles de séparation entre les mots. C'est le cas pour certains textes en latin classique, en grec ancien, en chinois, en japonais ou encore en vietnamien, pour ne citer que quelques exemples. Ce type d'écriture, appelé *scripto continua* est resté en usage en Occident jusqu'au X^e siècle environ. La **figure I.2** propose un exemple de document utilisant cette *écriture continue*. On peut attribuer l'introduction des espaces pour séparer les mots à des moines copistes irlandais aux alentours du VIII^e siècle.

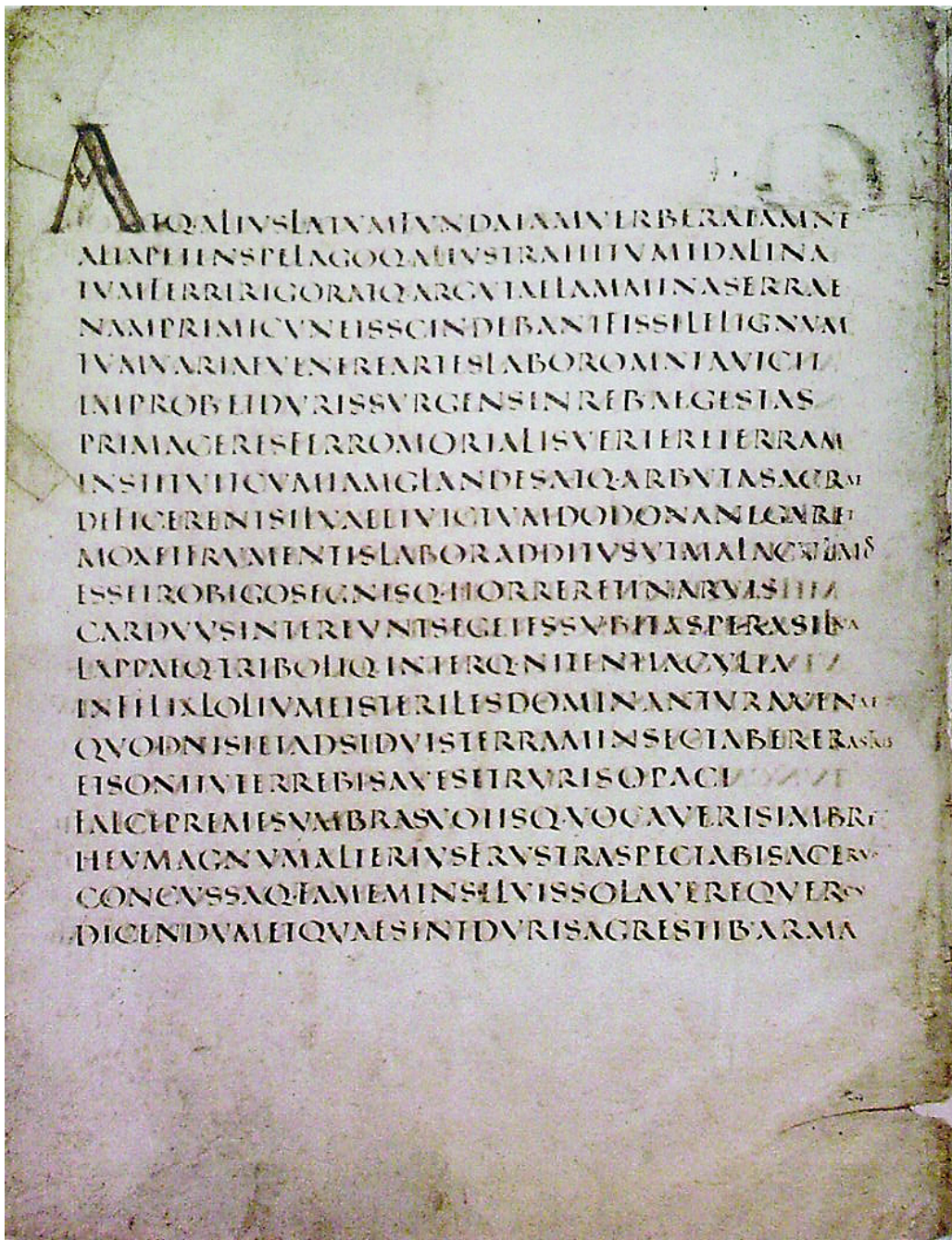


Figure I.2 Un extrait des *Géorgiques* de Virgile, en *scripto continua*. Pourrions-nous le segmenter en mots afin de le déchiffrer ?

I.C – Applications

Encore aujourd'hui, la segmentation d'un texte en mots a de nombreuses applications en traitement automatique des langues. Comme indiqué ci-dessus, il peut s'agir d'une première étape de prétraitement d'un texte en chinois ou en japonais. Le problème se pose également en français pour le traitement des *mots-dièses* (`#segmentationenmots`) ou en allemand pour la segmentation de mots composés (par exemple comment segmenter le mot *Rindfleischetikettierungsueberwachungsaufgabenuebertragungsgesetz* que l'on peut traduire par « loi sur le transfert des obligations de surveillance de l'étiquetage de la viande bovine » ?). Enfin, le problème est similaire lorsque l'on cherche à segmenter un texte en phrases (la notion de « mot » désignant alors une phrase).

Dans cette épreuve, à des fins d'illustration, nous reprendrons l'exemple ludique d'un texte en français dont on a égaré les espaces.

I.D – Formalisation

Considérons un alphabet Σ et soit $L \subseteq \Sigma^+$ un langage qui, intuitivement, va correspondre à l'ensemble des mots valides. Soit $t \in \Sigma^*$ un texte. On cherche une décomposition $t = u_1 u_2 \dots u_n$ avec, pour $i \in \llbracket 1, n \rrbracket$, $u_i \in L$, autrement dit, on se demande si $t \in L^*$. On peut se demander si une telle décomposition existe, combien il en

existe ou bien encore chercher à les trouver toutes. On peut également se donner une fonction de score $s : L \rightarrow \mathbb{R}$ qui associe un score à chaque mot et chercher la décomposition de score $s(u_1) + s(u_2) + \dots + s(u_n)$ maximal.

Il y a deux problèmes différents qui vont être traités dans deux parties distinctes.

1. Il nous faut tout d'abord définir la notion de *mot*, c'est-à-dire en fait le langage L , cette définition dépendant bien entendu de l'application envisagée. Dans le cadre de ce sujet, on souhaite disposer du langage L de tous les *mots* français. Une possibilité est de prendre pour L l'ensemble de tous les mots rencontrés dans un corpus de textes. En mémorisant également le nombre d'occurrences de ces mots, on pourra ensuite en déduire de plus une fonction de score. Ce premier problème est abordé dans la **partie II** qui sera à implémenter en utilisant le langage C.
2. Il nous faut ensuite réaliser la segmentation proprement dite, en supposant désormais disposer d'une fonction `est_mot : string -> bool` qui vérifie l'appartenance d'un mot $u \in \Sigma^*$ au langage L ou d'une fonction `score : string -> float` qui associe à un mot $u \in \Sigma^*$ son score $s(u)$. Ce deuxième problème est abordé dans la **partie III** qui sera à implémenter en utilisant le langage OCAML.

I.E – Description des données

Ce sujet est accompagné d'un ensemble de fichiers pour accompagner votre travail :

- un fichier `lexique.c` est à compléter pour la **partie II** ;
- un fichier `segmentation.ml` est à compléter pour la **partie III** ;
- le répertoire `data` contient un ensemble de fichiers de données d'exemples, sur lesquels vous devrez travailler, sans les modifier.

Le fichier `les_trois_mousquetaires.txt` contient le texte correspondant au roman *Les Trois Mousquetaires* d'Alexandre DUMAS à raison d'un mot par ligne. Ce que l'on appelle ici *mot* est une suite de caractères qui peut se trouver sur une de ces lignes. Ainsi « , » ou « . » sont des mots.

Les dix premiers mots de ce fichier sont données en **figure I.3**.

```
préface
dans
laquelle
il
est
établi
que
,
malgré
leurs
```

Figure I.3 Les dix premières lignes du fichier `les_trois_mousquetaires.txt`

De même, le fichier `les_miserables.txt` correspond aux cinq tomes du roman *Les Misérables* de Victor HUGO. Enfin, le fichier `la_comedie_humaine.txt` correspond aux douze premiers volumes de *La Comédie Humaine*, l'œuvre d'Honoré DE BALZAC.

On suppose dans tout ce sujet qu'aucun mot n'est de longueur strictement supérieure à 100. Les programmes pourront utiliser cette hypothèse sans la vérifier.

Le fichier `les_miserables.lex` contient tous les mots apparaissant dans le fichier `les_miserables.txt`, avec leur nombre d'occurrences. Ce fichier comporte une ligne par mot, chaque ligne étant constituée du mot et de son nombre d'occurrences séparés par une espace. La **figure I.4** présente un extrait de ce fichier qui indique que le mot `absolument` apparaît 60 fois dans le roman de HUGO.

```
absolu 39
absolue 14
absolues 1
absolument 60
absolus 1
```

Figure I.4 Extrait du fichier `les_miserables.lex`

Le fichier `les_miserables.big` contient tous les couples de mots consécutifs, appelés *bigrammes*, apparaissant dans le fichier `les_miserables.txt`, avec le nombre d'occurrences de chaque bigramme. Ce fichier comporte une ligne par mot, chaque ligne comportant les deux mots du bigramme et le nombre d'occurrences de ce bigramme, chacun séparé par une espace. La **figure I.5** présente un extrait de ce fichier qui indique que le mot `absolument` apparaît trois fois suivi d'une virgule, une fois suivi d'un point, une fois suivi du mot `autre`, etc.

```
absolument ! 1
absolument , 3
absolument . 1
absolument autre 1
absolument besoin 1
```

Figure I.5 Extrait du fichier `les_miserables.big`

L'objectif de la **partie II** est de générer les fichiers correspondants pour les deux autres textes `les_trois_mousquetaires.txt` et `la_comedie_humaine.txt`. Les deux fichiers `les_miserables.lex` et `les_miserables.big` ainsi que les fichiers générés dans la **partie II** seront utilisés dans la **partie III**.

II Constitution d'un lexique (en C)

Un fichier `lexique.c`, à compléter, est fourni pour cette partie.

II.A – Mots et nombre d'occurrences

Pour calculer le nombre d'occurrences de chaque mot dans un texte, nous allons utiliser un dictionnaire que l'on demande d'implémenter à l'aide d'un arbre binaire de recherche. On ne cherchera pas, du moins dans un premier temps, à équilibrer les arbres.

On utilisera la structure suivante pour implémenter un nœud d'un arbre binaire de recherche. Les clés correspondent aux mots et les valeurs associées à ces clés au nombre d'occurrences de ces mots dans le texte. On représente un arbre binaire de recherche par un pointeur sur sa racine, donc par un objet de type `node*`. Le pointeur `NULL` permet de représenter un arbre binaire de recherche vide.

```
struct node_s {
    char* key;
    int val;
    struct node_s* left;
    struct node_s* right;
};
typedef struct node_s node;
```

On prêtera une attention particulière à la gestion de la mémoire en C. En particulier, on adoptera dans les programmes une convention qui indique clairement, pour chaque structure de données, quelle partie du code est responsable des allocations et des libérations de mémoire.

Pour pouvoir implémenter un dictionnaire par un arbre binaire de recherche, il faut se donner un ordre total sur les clés. On se propose ici d'utiliser l'ordre lexicographique sur les chaînes de caractères. On comparera simplement les éléments de type `char` un par un avec l'opérateur `<`, sans se préoccuper de questions d'encodage.

On rappelle que l'on peut librement utiliser les fonctions de la bibliothèque standard C et en consulter la documentation, en particulier les fonctions de `string.h`, notamment `strlen`, `strcpy` et `strcat`. On s'interdit cependant l'utilisation de `strcmp`.

Q 1. 📄 Écrire une fonction de prototype `int lexcmp(char* s1, char* s2)` qui compare deux chaînes de caractères `s1` et `s2` passées en paramètres. Si la première chaîne est strictement inférieure à la deuxième pour l'ordre lexicographique, cette fonction renvoie la valeur `-1` ; si les deux chaînes sont égales, cette fonction renvoie la valeur `0` ; sinon, donc si la première chaîne est strictement supérieure à la deuxième pour l'ordre lexicographique, cette fonction renvoie la valeur `1`.

Q 2. 📄 Décrire le fonctionnement de votre fonction, en justifier la correction et en donner sa complexité temporelle et spatiale.

Le fichier fourni `lexique.c` propose une implémentation d'une fonction `print_bst` que vous pouvez utiliser pour tester vos futures fonctions. Une implémentation de deux fonctions `delete_min` et `delete` vous est également proposée.

Q 3. 📖 Décrire ce que réalisent les deux fonctions `delete_min` et `delete`, en explicitant leur fonctionnement. Détailler en particulier les mécanismes de gestion de la mémoire mis en œuvre dans ces deux fonctions.

Q 4. 📖 Implémenter la ou les fonctions qui vous semblent nécessaires pour pouvoir manipuler un arbre binaire de recherche, dans l'optique stricte de son utilisation pour construire un lexique associant à chaque mot d'un texte son nombre d'occurrences.

Q 5. 📖 Donner le prototype des fonctions réalisées.

Q 6. 📖 📖 Proposer un jeu de test permettant de vérifier le bon fonctionnement de vos fonctions.

On rappelle l'existence des fonctions de manipulation de fichiers définies dans l'en-tête `stdio.h` : `fopen`, `fclose`, `fscanf`, `fprintf`. Si `file` est un descripteur de fichier de type `FILE*` ouvert avec `fopen`, si `line` est un pointeur de type `char*` pointant vers une zone de mémoire déjà allouée et de taille suffisante, alors l'appel `fscanf(file, "%s", line)` ; lit une ligne entière¹ du fichier et la stocke dans `line`. Le caractère de retour à la ligne est lu par `fscanf` mais n'est pas stocké dans `line`. La fonction `fscanf` renvoie la valeur 1 en cas de succès. Lorsqu'elle atteint la fin du fichier, elle renvoie l'entier défini par la constante `EOF`.

Q 7. 📖 Écrire un programme complet permettant de constituer un lexique comportant exactement tous les mots apparaissant dans un texte (comportant un mot par ligne) avec leur nombre d'occurrences. Appliquer ce programme au texte d'Alexandre DUMAS et enregistrer le lexique dans `les_trois_mousquetaires.lex`, un nouveau fichier comportant une ligne par mot, chaque ligne comportant le mot et son nombre d'occurrences séparés par une espace, comme indiqué dans la **section I.E**. Faire de même avec le texte de *La Comédie Humaine* de BALZAC.

On pourra comparer le lexique obtenu pour le fichier `les_miserables.txt` avec celui qui vous est proposé. *Mais attention à ne pas écraser le lexique qui vous est fourni par un fichier incorrect !*

Q 8. 📖 Répondre aux questions suivantes pour les fichiers des deux textes fournis `les_trois_mousquetaires.txt` et `la_comedie_humaine.txt` :

- Combien il y a-t-il de mots différents ?
- Quel est le mot le plus fréquent et quel est son nombre d'occurrences ?
- Combien de mots apparaissent exactement 10 fois ?
- Quelle est la hauteur de l'arbre binaire de recherche calculé ?

Q 9. 📖 Quelle est la complexité temporelle de la constitution de ce lexique dans le pire cas ? Décrire une technique permettant d'équilibrer un arbre binaire de recherche. Quelle serait alors la complexité temporelle de la constitution du lexique dans le pire cas ? En pratique, les arbres obtenus ici sans équilibrage vous semblent-ils équilibrés ?

II.B – Bigrammes

En fonction de votre avancement, il peut être judicieux de commencer par traiter la partie suivante avant de revenir à cette question.

Un *bigramme* est une suite de deux mots consécutifs dans le texte.

Q 10. 📖 Écrire un programme en C permettant de constituer la liste des bigrammes apparaissant dans un texte avec le nombre d'occurrences de chaque bigramme. Appliquer ce programme au texte d'Alexandre DUMAS et enregistrer ces bigrammes dans un nouveau fichier `les_trois_mousquetaires.big` comportant une ligne par mot, chaque ligne comportant les deux mots du bigramme et le nombre d'occurrences, chacun séparés par une espace. Faire de même avec l'œuvre de BALZAC.

¹ Du moins dans le cas présent, les lignes du fichier ne contenant pas de caractères d'espacement.

III Segmentation d'un texte (en OCAML)

Un fichier `segmentation.ml` est fourni pour cette partie. Ce fichier propose une fonction `verifie_format` qui vérifie qu'un fichier de lexique est au format attendu. Vous pouvez l'utiliser pour vérifier que votre fichier `les_trois_mousquetaires.lex` est bien formé.

Considérons le mot `beaucoupdevin` et le langage intuitif correspondant à l'ensemble des mots français. La segmentation *naturelle* semble être `beaucoup de vin`, mais d'autres segmentations sont possibles, comme par exemple `beau coup de vin` ou `beaucoup devin`. Le but de cette partie est de déterminer, pour un mot donné et un vocabulaire correspondant à un lexique, s'il existe une telle segmentation, combien il en existe ou encore de déterminer une « meilleure » segmentation, c'est-à-dire une segmentation de score maximal.


Pour simplifier la mise au point du programme, on pourra considérer le langage $L_{ex} = \{a, ab, aba, bb\}$ sur $\Sigma = \{a, b\}$. Le mot `babbab` n'admet aucune segmentation suivant L alors que le mot `abaaba` en admet 4.

III.A – Lecture du lexique

On représente un lexique en OCAML à l'aide du type suivant :


```
type lexique = (string, int) Hashtbl.t
```

La table de hachage `dico_ex` du fichier `segmentation.ml` contient un exemple de lexique correspondant au langage L_{ex} pour les occurrences $(a, 1)$, $(ab, 1)$, $(aba, 2)$, $(bb, 1)$. Vous pourrez utiliser ce dictionnaire pour tester vos fonctions.

Q 11.  Écrire un programme en OCAML qui lit un lexique à partir d'un fichier texte, comme par exemple `les_miserables.lex`, et qui enregistre son contenu dans une table de hachage de type `lexique`. On pourra s'inspirer de la fonction `verifie_format` comme modèle pour lire les données et consulter la documentation du module `Hashtbl`.

Q 12.  Pour le fichier `les_miserables.lex` :

- Combien il y a-t-il de mots différents ?
- Quel est la longueur du mot le plus long ?
- Combien de mots apparaissent exactement une fois ?


Q 13.  Écrire une fonction `est_mot : lexique -> string -> bool` qui vérifie si une chaîne de caractères est un mot valide pour un lexique et une fonction `score : lexique -> string -> float` qui renvoie, pour le moment, le logarithme du nombre d'occurrences d'un mot dans le lexique s'il est présent et $-\infty$ sinon. On pourra utiliser la fonction `log` et la constante `neg_infinity`.


On souhaite dans un premier temps déterminer une segmentation possible, s'il en existe au moins une, d'un texte $t \in \Sigma^*$ suivant le langage $L \subseteq \Sigma^+$, le langage L étant défini par la fonction `est_mot`, c'est-à-dire que $L = \{u \in \Sigma^* \mid \text{est_mot}(u) = \text{true}\}$.

III.B – Une approche gloutonne

Q 14.  Que pensez-vous d'une approche gloutonne pour résoudre ce problème ? Justifier.

Dans certaines applications, on peut supposer qu'aucun mot du langage L n'est préfixe d'un autre mot de ce langage. On appelle un tel langage L un *code préfixe*. Ce n'est pas le cas en français puisque par exemple `beau` est préfixe de `beaucoup`.

Q 15.  En supposant que le langage L des mots possibles est un code préfixe, proposer une structure de données adaptée et décrire le fonctionnement d'un algorithme pour résoudre le problème de la segmentation d'un texte en mots. *On ne demande pas d'implémenter cette approche en OCAML, mais simplement de présenter vos idées à votre examinatrice ou examinateur.*

Q 16.  Le codage de Huffman utilise un code préfixe. Rappeler le principe et l'intérêt de ce codage, expliquer sommairement l'algorithme de construction de l'arbre de Huffman et indiquer sa complexité.

III.C – Une approche par retour sur trace

Remarquons que ce problème de segmentation se décompose assez naturellement récursivement en sous-problèmes. Si $t \in \Sigma^*$ est décomposable suivant L , alors il existe un préfixe $u \in L$ (qui est le premier mot d'une décomposition) et $t' \in \Sigma^*$ (ce qu'il reste à décomposer) tels que $t = ut'$. Il reste ensuite à décomposer t' . En fonction du choix du préfixe u , cette décomposition peut être possible ou non, ce qui suggère un algorithme de

retour sur trace (*backtracking* en anglais). On peut tester successivement tous les préfixes $u \in L$ de t candidats et essayer de poursuivre la décomposition sur ce qu'il reste. Si l'on réussit à obtenir une décomposition, on peut s'arrêter avec succès, sinon on *revient en arrière* sur le préfixe suivant.

Par exemple, considérons le mot *abab* pour le langage L_{ex} . Le premier préfixe appartenant à L_{ex} est a . Il reste à segmenter *bab* pour lequel aucun préfixe ne convient et qui n'est donc pas segmentable. On revient en arrière et on considère le préfixe suivant appartenant à L_{ex} de *abab* qui est ab pour lequel il reste à segmenter *ab*. Pour segmenter *ab*, on tente le préfixe a de *ab*, qui ne convient pas puisque b n'est pas segmentable. On tente le préfixe ab de *ab* qui permet de segmenter entièrement le mot *ab*. On peut s'arrêter avec succès et on a trouvé la segmentation *ab ab*.

Q 17. 📖 Décrire votre idée d'implémentation, ainsi que les structures de données que vous pensez utiliser à votre examinatrice ou examinateur avant de vous lancer dans l'implémentation. Comment allez-vous mettre en œuvre la recherche de « tous les préfixes $u \in L$ de t » ?

Q 18. 🖨️ Implémenter cette stratégie de retour sur trace en OCAML. On pourra dans un premier temps chercher uniquement à savoir si une segmentation existe, avant de modifier la fonction pour renvoyer également une telle segmentation le cas échéant. On pourra utiliser le lexique correspondant au langage L_{ex} proposé avant de tester avec le lexique issu du fichier `les_miserables.lex`. *Remarque : il est normal d'obtenir une segmentation peu convaincante.*

Q 19. 🖨️ Proposer une modification de votre fonction pour dénombrer le nombre de segmentations possibles d'un texte t suivant un langage L .

Q 20. 🐿 Avec le lexique `les_miserables.lex` et le texte `laquelleestlameilleure`, combien existe-t-il de segmentations de ce texte ?

Q 21. 🐿 Quelle est la complexité de cette approche ? Détailler précisément le calcul de complexité effectué.

III.D – Amélioration de l'approche naïve

Q 22. 📖 🖨️ Expliquer en quoi l'approche précédente conduit à recalculer de nombreuses fois les mêmes sous-problèmes. Proposer une amélioration à cette approche et l'implémenter. Comment s'appelle ce paradigme de programmation ?

Q 23. 🐿 Avec le lexique `les_miserables.lex`, combien existe-t-il de segmentations du texte `texte_preface` que l'on trouve dans le fichier `segmentation.ml`, correspondant au chapeau de la préface du roman d'Alexandre DUMAS ? On donnera les six derniers chiffres uniquement.

III.E – Segmentation de score maximal

On souhaite maintenant trouver la meilleure segmentation possible, lorsqu'il en existe au moins une, selon la fonction de score `score`.

Q 24. 🖨️ Proposer une modification de votre fonction pour trouver la segmentation de score maximal d'un texte t suivant un langage L .

Q 25. 🐿 Avec le lexique `les_miserables.lex` quelle est la segmentation de score maximal obtenue pour le texte `laquelleestlameilleure` ? Quel est le score obtenu pour le texte de la préface `texte_preface` ? *Remarque : il est normal d'obtenir une segmentation peu convaincante.*

III.F – Utilisation d'une heuristique

La segmentation de score maximal obtenue n'est pas très satisfaisante ou du moins ne correspond pas à notre idée d'une « bonne » segmentation. On observe un biais vers l'utilisation de mots très courts lors de la segmentation. En effet, les mots courts ont tendance à être plus fréquents que les mots longs dans un texte, et surtout, le caractère additif du score $s(u_1) + s(u_2) + \dots + s(u_n)$, dont tous les termes sont positifs, encourage les segmentations comportant de nombreux facteurs.

Pour contrebalancer cet effet, nous allons introduire une *heuristique* et prendre en compte la longueur d'un mot. On ajoute au score le carré de la longueur du mot. On considère donc le score :

$$s(u) = s_{lex}(u) + |u|^2$$

où s_{lex} est le score lexical correspondant au logarithme du nombre d'occurrences de ce mots donné par le lexique.

Q 26. 📖 Avec cette nouvelle fonction de score et le lexique `les_miserables.lex` quelle est la meilleure segmentation obtenue pour le texte `laquelleestlameilleure`? Quel est le score obtenu pour le texte de la préface `texte_preface`? *Remarque : on obtient maintenant une segmentation convaincante.*

Q 27. 📖 Comparer, discuter et commenter les segmentations de score maximal obtenues pour le chapeau de la préface du roman d'Alexandre DUMAS `texte_preface` en utilisant les trois lexiques `les_trois_mousquetaires.lex`, `les_miserables.lex` et `la_comedie_humaine.lex`.

III.G – Bigrammes

On peut remarquer que le score d'une phrase ne dépend pour l'instant que de la fréquence (et de la longueur) *individuelle* de ses mots. Ainsi par exemple une phrase `on a un` obtient le même score qu'une phrase `a on un` alors que la première phrase semble bien plus naturelle en langue française. On peut préférer obtenir une segmentation dans laquelle la cooccurrence de deux mots consécutifs est forte.

On se propose d'introduire un modèle *bigramme* dans lequel on attribue en plus un score à chaque bigramme de mots. Comme pour le score *lexical* précédent, on va utiliser un score *bigramme* s_{big} comme le logarithme du nombre d'occurrences de ce bigramme dans un texte. Le score d'une *segmentation* sera donc :

$$s(u_1 u_2 \dots u_n) = \sum_{i=1}^n (s_{lex}(u_i) + |u_i|^2) + \sum_{i=1}^{n-1} s_{big}(u_i u_{i+1}).$$

Q 28. 📖 Implémenter cette approche en OCAML.

Q 29. 📖 Proposer des exemples qui montrent que cette approche peut se révéler meilleure ou moins bonne suivant les cas.

III.H – Mots inconnus

Il y a également un problème pour les mots *inconnus*. Par exemple, le mot *artagnan* apparaît 1870 fois dans *Les Trois Mousquetaires* mais n'apparaît pas dans le lexique issu du roman *Les Misérables*. On souhaite cependant être capable de segmenter les phrases contenant le nom de ce personnage.

Q 30. 📖 Proposer une solution à ce problème.

Remarques générales sur le sujet d'exemple

Le jury donne dans ce document un exemple de sujet qui pourrait être proposé pour l'épreuve pratique d'informatique en filière MPI. Ce sujet montre comment le programme des classes de MP2I et MPI peut être exploité dans le cadre d'une épreuve orale sur machine.

Le jury sera attentif aux remarques sur ce sujet 0 qui pourront lui être adressées à l'adresse tp-info.sujet0@concours-centrale-supelec.fr avant le 16 juin 2023, notamment par les enseignants de la filière.

Cette épreuve d'informatique est conçue comme une véritable épreuve orale, durant laquelle le jury s'attend à avoir de nombreux échanges avec les candidates et les candidats. Il ne s'agit *pas* d'une épreuve de programmation en autonomie complète sur la durée totale. L'épreuve se déroule dans une salle où plusieurs candidates ou candidats sont interrogés en même temps.

Le jury cherchera à évaluer en particulier les points suivants :

- connaissances théoriques sur les notions et les algorithmes des programmes de MP2I et MPI. Le jeu de sujets proposé par le jury permettra d'évaluer l'ensemble des programmes des deux années de formation ;
- maîtrise des traits de programmation des langages. L'ensemble des sujets permettra d'évaluer tous les langages de la filière MPI (C, OCAML, SQL). Chaque sujet séparément peut, selon la pertinence pour le thème abordé, utiliser un seul langage, ou comporter plusieurs parties à traiter chacune dans son propre langage imposé par le sujet, ou encore laisser explicitement certaines parties au libre choix des candidates et candidats ;

Pendant l'épreuve, le jury peut fournir des fichiers aux candidates et aux candidats :

- des fichiers sources, complets ou à compléter, en C, OCAML ou SQL ;
- des fichiers de données, dont le format sera soit explicitement décrit par l'énoncé (s'il est demandé d'écrire un programme qui lit les données), soit lisible par des fonctions pré-écrites et fournies par le sujet ;
- des scripts simplifiant la compilation des programmes, dont l'usage sera documenté par le sujet. En l'absence de tels scripts, les candidates et les candidats doivent savoir invoquer les compilateurs C et OCAML pour compiler leurs programmes, si besoin en consultant la documentation ;
- de la documentation (`man`, copie de manuels des langages).