LANGAGES DE PROGRAMMATION
IFT-3000
HIVER 2017

Travail pratique 2 - TRAVAIL INDIVIDUEL OU EN ÉQUIPE

Pondération de la note finale : 14.5%

À remettre, avant 17h, le vendredi 28 avril 2017

1 Énoncé

Ce travail pratique consiste à implanter différentes fonctions qui entrent en compte dans un contexte de correction orthographique. Vous devez utiliser les notions que nous avons vues dans le cours en faisant des manipulations de base sur un système qui est implanté en utilisant différents types de structures de données : listes, tables de hachage, etc.

Lien avec le TP1: Il s'agit d'une suite logique du TP1 puisque dans ce dernier, on complétait automatiquement des préfixes saisis par l'utilisateur en lui suggérant des mots existants dans un dictionnaire donné, mots triés de manière croissante selon leur taille; dans le TP2, on aimerait proposer à l'utilisateur, à partir des mots déjà saisis, ou à partir de quelques caractères déjà saisis (même s'ils ne forment aucun préfixe d'un mot existant dans un dictionnaire), la liste des mots les plus probables qu'il aurait l'intention de saisir; ainsi, en plus de lui proposer des mots qui corrigeraient la séquence de caractères qu'il aura saisie, le système tient compte des mots déjà saisis, le *contexte*, afin de raffiner davantage sa liste des meilleurs candidats (mots) à proposer à l'utilisateur. Évidemment, lorsque cette séquence forme un préfixe de mots existants, il sera toujours possible d'utiliser la fonction «liste_par_prefixe» du TP1 pour énumérer les mots possiblement candidats; mais la prise en compte du *contexte* permettrait de mieux trier ces mots (relativement au simple tri basé sur la taille des mots, utilisé dans le TP1).

En résumé, dans le TP2, il sera possible de :

1. de prendre en entrée un texte, de détecter les fautes d'orthographes et de les corriger automatiquement;
2. de proposer, à la volée, comme dans le cadre de l'application proposée dans le fichier «gui.ml» du TP1, des mots à l'utilisateur selon ce qui a été saisi par l'utilisateur (en tenant compte des mots déjà saisis), même s'il ne correspond à aucun préfixe de mots existants (dans un dictionnaire).

1.1 Définitions utiles

Cette section contient un ensemble de définitions utiles dans le contexte de la correction d'orthographe et du traitement du langage naturelle en général.

- Corpus: ensemble de textes tirés d'un domaine d'application donné et servant à estimer des probabilités de mots. Le nombre total de mots dans un corpus est noté $|N|$ et le nombre total de mots distincts de ce corpus est noté $|V|$. V est aussi désigné comme le vocabulaire pour le domaine d'application d'intérêt.
- Mot: un mot d'un corpus est un ensemble de caractères délimité par des espaces ou des ponctuations. Soit w un mot de longueur $|w|$, la notation $w = w_1 w_2 ... w_{|w|}$ veut simplement dire que le mot est composé des caractères $w_1, w_2, ..., et\ w_{|w|}$. En écrivant w_i on désigne le caractère à la ième position d'un mot w.
- n-gramme: ce sont des séquences de n mots dans un texte. Par exemple, les 1-gramme (ou unigramme) de la phrase[1] " i love this color " sont: "i", "love", "this", "color"; ces 2-gramme (ou bi-gramme) sont: "i love", "love this", "this color".

[1] Il est d'usage de considérer en minuscule tous les mots traités.

1.2 Correction orthographique - Présentation

La correction d'orthographe consiste principalement à détecter les fautes d'orthographes puis à les corriger automatiquement, ou en suggérant des corrections possibles. On retrouve principalement deux types de fautes d'orthographes: i- les fautes menant à des mots inexistants dans la langue écrite (ex: «pait» au lieu de «paix») et ii- les fautes menant à des mots existants (ex: «voie» au lieu de «voix»). Il est donc évident que la correction orthographique requiert obligatoirement un dictionnaire de mots bien orthographiés, appelé aussi vocabulaire. Le vocabulaire sera appelé V dans le reste du document.

Dans le contexte de ce travail, nous allons nous intéresser uniquement aux fautes du premier type (i) mais tout le travail effectué pourra facilement être extensible aux fautes du second type. Aussi, nous allons uniquement travailler avec la langue anglaise mais il sera aussi possible, en changeant des jeux de données que nous allons utiliser, de faire la correction dans n'importe quel autre langue.

La correction des fautes de mots inexistants se déroule en trois phases. Nous allons passer au travers de ces trois phases en prenant pour exemple de texte à corriger la phrase suivante "i want thi color.". Dans notre exemple, considérons le vocabulaire $V = \{i, a, an, want, the, this, color, shoe\}$

1. **détection de la faute**: C'est la partie la plus simple du processus de correction. Un mot de votre texte à corriger est une faute, s'il n'appartient pas au dictionnaire (ie ne fait pas partie du vocabulaire). Dans la suite de ce document, x désignera systématiquement un mot en faute. Dans le contexte de notre exemple, le seul mot qui n'appartient pas au dictionnaire est "thi"; donc c'est une faute à corriger.

2. **génération des candidats possibles**: Étant donné V, cette étape consiste à y trouver les mots similaires au mot mal orthographié x. On dira souvent que deux mots sont similaires lorsque le nombre d'opérations (insertion, suppression, substitution, transposition), pour passer d'un mot à l'autre, est inférieur à 2. Ce nombre d'opérations sera calculé grâce à la *distance d'édition* de Damerau-Levenshtein. Les détails algorithmiques sur cette distance sont fournis à l'adresse suivante https://fr.wikipedia.org/wiki/Distance_de_Damerau-Levenshtein. Il faudrait donc appliquer l'algorithme de la distance d'édition à tous les mots de V, couplés avec le mot fautif x. L'ensemble des mots ayant une distance inférieure ou égale à 2 avec la faute sera noté C et constituera l'ensemble des candidats à considérer pour les étapes subséquentes de la correction. Soit d, la fonction qui calcule la distance. Dans l'exemple ci-dessus, on aura à appeler cette fonction avec les mots de notre petit vocabulaire V et la faute x = "thi", et on obtiendra:

   - $d(i, thi) = 2$,
   - $d(a, thi) = 3$,
   - $d(an, thi) = 3$,
   - $d(the, thi) = 1$,
   - $d(this, thi) = 1$,
   - $d(color, thi) = 5$,
   - $d(shoe, thi) = 3$

   De ces résultats, on pourra conclure que le sous-ensemble des mots candidats est $C = \{i, this, the\}$. En effet, pour passer du mot "i" au mot "thi", seules 2 opérations sont nécessaires: insertion du caractère "h" puis insertion du caractère "t". Pour passer de "this" à "thi", seule une opération est nécessaire: la suppression de "s". Et pour passer de "the" à "thi", il suffit d'une opération qui consiste à substituer "e" par "i".

   En situation réelle, le vocabulaire est beaucoup plus imposant, le calcul de la distance pour tous les mots du vocabulaire est donc à éviter (engendre trop de temps de calcul). Une idée pour diminuer le nombre de calcul est de considérer seulement les mots dont la taille est comprise entre $|x|-2$, et $|x|+2$ (en effet, au delà de cette intervalle, on peut facilement considérer que la distance d'édition sera supérieure à 2).

3. **choix du meilleur candidat**: Étant donné l'ensemble des candidats C, cette étape utilise un modèle probabiliste pour trouver le meilleur candidat (ie le mot correct). Soit w^* le mot correct, le modèle probabiliste que nous utiliserons dans ce travail énonce que:

$$w^* = \underset{w \in C}{\operatorname{argmax}} \, P(w|x) \tag{1}$$

   C'est à dire que dans l'ensemble des mots candidats C, le mot correct w est celui ayant la plus forte (argmax) probabilité d'être là sachant qu'on a observé le mot fautif ($P(w|x)$). En utilisant la règle de Bayes, cette dernière équation revient à

$$w^* = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{modèle de canal}} \times \underbrace{P(w)}_{\text{modèle de langue}} \tag{2}$$

Sous cette nouvelle formulation on se rend compte que l'on a besoin de savoir calculer (estimer) deux probabilités pour résoudre le problème: $P(x|w)$ et $P(w)$. Le calcul de ces probabilités est détaillé dans les 2 sections qui suivent. Pour l'instant, supposons que nous savons comment calculer ces probabilités. Pour chacun des mots candidats, il faut donc les calculer, puis les multiplier. Le mot correct à proposer à l'utilisateur est celui qui aura la plus grande valeur pour ce calcul (multiplication de 2 calculs de probabilités).

1.2.1 Modèle de canal

Étant donné le mot fautif x, et un mot du candidat w, ce modèle probabiliste permet d'estimer la probabilité qu'un utilisateur puisse écrire le mot fautif x alors qu'il avait en tête le mot w. Cette probabilité est le produit de toutes les probabilités des opérations (fautes) qu'il faut faire pour passer de w à x. En gros, pour calculer la probabilité que l'on recherche, nous devons :

1. d'abord, trouver des fautes; l'algorithme de la distance d'édition, légèrement modifié, pourra fournir, en plus de la distance entre un mot $source$ et un mot $cible$, la liste des opérations à effectuer pour passer du premier au deuxième mot. Cette liste correspond à la liste des fautes qu'on commettrait lorsqu'on voudra écrire un mot w (source) et qu'on aboutira, à la place, à un mot x (cible);

   Pour les mots candidats de notre exemple ($C = \{i, this, the\}$), les fautes pour passer au mot mal orthographié "thi" sont:

   - avec i: $\{\text{INS}(t, \epsilon), \text{INS}(h, \epsilon)\}$
   - avec $this$: $\{\text{DEL}(s, i)\}$
   - avec the: $\{\text{SUB}(e, i)\}$

   $\text{INS}(t, \epsilon)$ et $\text{INS}(h, \epsilon)$ consistent respectivement à insérer les caractères t et h après le caractère spécial ϵ marquant la début du mot i; $\text{DEL}(s, i)$ consiste à supprimer le caractère s qui suit le caractère i; $\text{SUB}(e, i)$ est l'opération qui consiste à substituer, dans the, le caractère e par le caractère i.

2. ensuite, multiplier entre elles les probabilités de chacune de ces fautes afin d'obtenir le résultat final.

Pour le calcul de ces probabilités, vous aurez besoin de certaines matrices, appelées *matrices de confusion*, qui donnent le nombre de fois qu'une certaine faute (correspondant à une suppression, une insertion, une substitution ou une transposition) a été observée dans un ensemble de données. Ces matrices sont habituellement construites à partir de jeux de données répertoriant les fautes communes dans une langue. Dans le cadre de ce travail, nous allons utiliser l'ensemble de données fourni à l'adresse https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines et disponible dans le fichier «misspell.txt» (fourni dans le cadre de ce travail pratique). À partir de ces données, il est possible de construire une matrice de confusion (nxn caractères de l'alphabet (typiquement 26x26)), pour chaque type de fautes, tel que chacune des entrées de ces matrices représente une faute et nous donne son compte (le nombre de fois que cette faute apparaîtrait dans le jeu de données considéré) :

- del[a , b]: cette entrée de la matrice del comptabilise le nombre de fois que voulant "ab", on aura écrit "a";
- ins[a , b]: cette entrée de la matrice ins comptabilise le nombre de fois que voulant "a", on aura écrit "ab";
- sub[a , b]: cette entrée de la matrice sub comptabilise le nombre de fois que voulant "a", on aura écrit "b";
- $transp$[a , b]: cette entrée de la matrice $transp$ comptabilise le nombre de fois que voulant "ab", on aura écrit "ba".

S'il existe seulement une faute faite dans le mot w, afin d'obtenir le mot fautif x, on peut calculer la probabilité $P(x|w)$ comme suit (K étant le nombre de caractères ou de séquences de deux caractères successifs présents dans les mots correctement orthographiés du jeu de données («misspell.txt»)) :

$$P(x|w) = P(faute) = \begin{cases} \dfrac{del[w_{i-1}, w_i] + 1}{compte(w_{i-1}\, w_i) + |K|} & \textit{si faute de suppression de la forme DEL}(w_i, w_{i-1}) \\ \dfrac{ins[w_{i-1}, x_i] + 1}{compte(w_{i-1}) + |K|} & \textit{si faute d'insertion de la forme INS}(x_i, w_{i-1}) \\ \dfrac{sub[x_i, w_i] + 1}{compte(w_i) + |K|} & \textit{si faute de substitution de la forme SUB}(w_i, x_i) \\ \dfrac{transp[w_i, w_{i+1}] + 1}{compte(w_i\, w_{i+1}) + |K|} & \textit{si faute de transposition de la forme TRANSP}(w_i, w_{i+1}) \end{cases} \tag{3}$$

S'il en existe plusieurs, $n\ fautes$, on calcule la probabilité de chacune des fautes et on multiplie le tout, comme suit:

$$P(x|w) = P(faute_1) \times P(faute_2) \times ... \times P(faute_n) \tag{4}$$

Notons que :

- $compte(w_k)$ correspond au nombre de caractère w_k que l'on retrouve dans un jeu de données (comme celui référé précédemment; plus précisément, au niveau des mots correctement orthographiés) répertoriant les fautes communes dans une langue donnée;
- $compte(w_k w'_k)$ correspond au nombre de séquences des 2 caractères successifs $w_k w'_k$ que l'on retrouve dans ce même jeu de données (et de même, au niveau des mots correctement orthographiés).

1.2.2 Modèle de langue

Ce modèle probabiliste permet d'estimer la probabilité d'un mot (avec ou sans prise en compte du contexte qui l'entoure). Cette probabilité est estimée à partir d'un corpus d'entraînement.

Dans cette section, nous raffinons l'équation 2, présentée dans la section précédente à la page 2, afin de tenir compte du contexte qui entoure le mot w considéré.

Cas unigramme : pas de contexte Lorsqu'il n'y a pas de prise en compte de contexte, en se référant à l'équation 2, au niveau du modèle de langue, le calcul du meilleur candidat requiert le calcul de la probabilité $P(w)$; on parle de probabilité unigramme et elle est calculée comme suit :

$$P(w) = \frac{Compte(w) + 1}{|N| + |V|} \tag{5}$$

$Compte(w)$ correspond au nombre de fois qu'un mot w apparaît dans un corpus donné.

Cas bi-gramme : contexte qui comprend le mot u précédent En présence d'un contexte contenant un mot, on parle de probabilité bi-gramme. Ainsi, en considérant un mot u qui précède un mot w, dans l'équation 2, à la place de $P(w)$, il s'agit de calculer la probabilité $P(w|u)$; celle-ci se fait comme suit :

$$P(w|u) = \frac{Compte(uw) + 1}{Compte(u) + |V|} \tag{6}$$

$Compte(ww')$ correspond au nombre de fois que la séquence de mots ww' apparaît dans le corpus.

Cas bi-gramme : contexte qui comprend le mot v suivant De même, pour un mot v qui suivrait un mot w, dans l'équation 2, en plus de $P(w)$, il s'agit de calculer la probabilité $P(v|w)$; celle-ci se fait comme suit :

$$P(v|w) = \frac{Compte(wv) + 1}{Compte(w) + |V|} \tag{7}$$

Cas tri-gramme : contexte qui comprend le mot précédent u et le mot suivant v Dans le cadre de ce travail pratique, nous nous limitons à des probabilités bi-gramme; ainsi, le calcul de la probabilité tri-gramme qui tient compte à la fois du mot précédent et du mot suivant sera considérée comme une approximation utilisant des probabilités bi-gramme.

En considérant un mot u qui précède un mot w, et un mot v qui le succède, dans l'équation 2, à la place de $P(w)$, il s'agit de calculer la probabilité $P(w|u)$ et $P(v|w)$.

En résumé, pour le calcul du meilleur candidat :

- cas unigramme : $w^* = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{modèle de canal}} \times \underbrace{P(w)}_{\text{modèle de langue}}$
- cas bi-gramme (mot u précédent) : $w^* = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{modèle de canal}} \times \underbrace{P(w|u)}_{\text{modèle de langue}}$
- cas bi-gramme (mot v suivant) : $w^* = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{modèle de canal}} \times \underbrace{P(w) \times P(v|w)}_{\text{modèle de langue}}$
- cas tri-gramme (u et v) : $w^* = \underset{w \in C}{\operatorname{argmax}} \underbrace{P(x|w)}_{\text{modèle de canal}} \times \underbrace{P(w|u) \times P(v|w)}_{\text{modèle de langue}}$

Exemple : Considérons, comme exemple, la phrase «The symptoms include body aches.» :

- le mot w = «The» pourrait être analysé sans tenir compte du contexte ou de l'historique (probabilité unigramme); à ce moment-là, la probabilité $P(w)$ correspond au nombre d'occurrences du mot dans le corpus, plus 1, divisé par le nombre total de mots ainsi que du nombre de mots distincts du corpus;
- w pourrait aussi être analysé en tenant compte du mot qui le précède (probabilité bigramme), à savoir le mot de début de phrase (mot spécial noté «<s>»); à ce moment-là, à la place de $P(w)$, le calcul de la probabilité $P(w|<s>)$, c'est-à-dire la probabilité d'avoir le mot w en considérant qu'on a déjà identifié le mot qui le précède <s>, est le nombre de fois que mot w apparaît en début de ligne dans le corpus, plus 1, divisé par le nombre de fois que <s> apparaît dans le corpus, soit le nombre de lignes dans le corpus, plus le nombre de mots distincts du corpus;
- il en est de même si on considère le mot qui suit w, soit le mot v = «symptoms»; en plus de $P(w)$, $P(v|w)$ est calculée en suivant le même raisonnement;
- finalement, on peut considérer l'analyse du mot w = «The», en tenant à la fois compte du mot «<s>» qui le précède et du mot u qui le suit; dans le cas, il faut considérer $P(w|<s>)$ et $P(v|w)$.

1.2.3 En résumé

Dans le cadre de ce travail pratique, l'objectif est de corriger automatiquement un texte écrit dans une langue donnée (nous utiliserons par défaut la langue anglaise). À cette fin, comme précisé en introduction, nous suivons 3 étapes :

1. pour chaque mot analysé, on teste s'il est correct ou pas en le cherchant dans un dictionnaire donné; s'il est correctement orthographié, on passe au mot suivant; sinon, on poursuit avec les étapes 2 et 3;
2. étant donné le mot mal orthographié, grâce à une fonction de calcul de distance entre mots, on détermine un ensemble de mots du dictionnaire qui s'en rapprochent le plus, i.e. qui sont distants d'une valeur inférieure à 2; à ce niveau, on dispose, au final, d'un ensemble de mots candidats;
3. la dernière étape consiste à sélectionner, parmi tous les candidats retenus, celui qui serait le plus approprié pour correspondre au mot correctement orthographié; à cette fin, nous utilisons un modèle probabiliste à 2 niveaux :
   (a) en tenant compte du nombre de fois que certains types d'erreurs apparaissent habituellement dans une langue donnée; à ce niveau, nous utilisons un jeu de données permettant de déterminer ce type de statistiques; ce jeu de données (premier fichier) comprend des mots correctement orthographié ainsi que les habituels mots mal orthographiés qui leurs correspondent;
   (b) en tenant compte du nombre d'occurrences de chaque mot candidat, ainsi que le nombre d'occurrences de paires de mots (qui incluent chacun le mot candidat considéré), dans des textes écrits dans cette langue; à ce niveau, on utilise en général un corpus (deuxième fichier) composé d'un ensemble usuelle de phrases d'un domaine d'application précis.

Notons que pour le dictionnaire, il est d'usage de considérer la liste de tous les mots distincts du corpus choisi (on suppose, dans ce cas, que le corpus ne comprend pas de mots mal orthographiés); il est cependant possible de considérer un troisième fichier qui correspondrait à un dictionnaire comprenant une liste assez exhaustive des mots d'une langue donnée. Dans le cadre de ce travail, le dictionnaire correspondra aux mots distincts du corpus.

Remarque On comprend que la qualité des réponses fournies par les fonctions du Tp, en particulier celles qui corrigent automatiquement un texte donné, ou celles qui proposent des mots candidats relativement à un texte en cours de saisi, dépend de la qualité du corpus considéré, ainsi que du jeu de données répertoriant les fautes communes dans une langue donnée. Aussi, cette qualité dépend de la manière avec laquelle on considère l'analyse du texte en entrée, et plus précisément, l'identification des phrases (on parle de *segmentation de texte*), des mots qui les composent, des séparateurs de mots et de phrases, etc. (on parle de *normalisation de texte*) Dans le cadre du Tp, qui se veut être avant tout une introduction au domaine de la correction automatique, ainsi de l'analyse n-gramme, on se limite à des phrases délimitées par des fins de ligne, et à des mots formés de caractères alphabétiques (ainsi que de l'apostrophe) : tous les autres caractères sont tout simplement ignorés (ponctuation, etc.). Évidemment, une fois le Tp réalisé, il vous sera possible de l'étendre à ... l'infini! (peut-être pas autant que Google qui considérait, déjà en 2006, comme corpus une bonne partie du contenu d'Internet[2]!) Microsoft est de la partie aussi (depuis 2010) [3].

[2] https://research.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.
[3] https://blogs.msdn.microsoft.com/webngram/2010/10/04/language-modeling-101/.

2 Signatures

La signature des principales fonctions à implanter dans le cadre de ce travail pratique sont :

```
module Spellc :
  sig
    ...
    val liste_words : string -> string list list
    val wwpf_frequence : string -> (string, int) H.t * (string * string, int) H.t * int * int
    val liste_words' : string -> (string * string list) list
    val word_pair_list : string -> (string * string) list
    val gen_matrix: string -> int array array * int array array * int array array * int array array
    val c_cc_gen : string -> string list
    val gen_table : string -> (string, int) H.t
    val p_fault : operation -> float
    val prob_xw : operation list -> float
    val prob_uwv : ?u:string -> ?v:string -> string -> float
    val find_candidates : ?k:int -> string -> (string * operation list) list
    val best_candidate: ?u:string ->?v:string -> string -> (string * operation list) list -> string
    val tri_gramme : string list -> (string * string * string) list
    val revise_lwords : ?k:int -> string list -> string list
    val find_cand_aux : ?k:int -> ?u:string -> string -> (string * float) list
   ...
  end
```

Dans la section suivante, nous donnons plus de précision sur ces fonctions.

3 Implantation

Notez que le fichier «spellc.ml» est accompagné par un fichier «utiles.ml» définissant un module *Utiles* qui comprend un ensemble de fonctions utiles pour l'implantation des fonctions du travail pratique.

Par ailleurs, dans les sections qui suivent, nous détaillons les différentes fonctions à implanter.

3.1 Fonctions liées au corpus

Les 2 fonctions suivantes, que vous devez compléter, sont liées au corpus («corpus.txt») : identification de la liste des mots formant chaque ligne d'un fichier donné, et calcul de la fréquence (table de hachage *wf*) de tous les mots du fichier (noté N dans la section 1.1, et noté «_N» dans l'implantation), ainsi que la fréquence de toutes les paires de mots successifs (en tenant compte des mots spéciaux de début et de fin de phrase) (table de hachage *wpf*), et calcul du nombre de mots différents (noté V, pour vocabulaire, dans la section 1.1, et noté «_V» dans l'implantation).

liste_words Cette fonction lit les lignes d'un fichier :

- pour chaque ligne, elle transforme le tout en minuscule, puis appelle la fonction «split_line» pour déterminer la liste des mots formant une phrase :
  - si la liste est vide, elle l'ignore;
  - sinon, elle les retient parmi les éléments du résultat;
- elle retourne une liste de listes de mots correspondants aux listes des mots des phrases du fichier traité.

Remarques :

- l'ordre des lignes du fichier traité n'importe pas; par contre, pour chaque ligne traité, l'ordre des mots importe;
- dans le cadre de ce travail, une phrase correspond à une ligne d'un fichier traité; dans la réalité, une ligne d'un fichier traité peut comprendre plusieurs phrases délimités par un séparateur de phrases (point, point d'interrogation, etc.).

Exemples :

```
# liste_words "vide.txt";;
- : string list list = []
```

```
# liste_words "corpus.txt";;
- : string list list =
[["back"; "to"; "the"; "future"];
["i"; "hope"; "to"; "be"; "in"; "the"; "future"];
["the"; "objective"; "is"; "to"; "realize"; "my"; "dreams"];
["i"; "hope"; "to"; "realize"; "that"; "dream"];
["i"; "have"; "a"; "dream"]]
```

wwpf_frequence Pour un fichier donné, la fonction retourne :

- une table de hachage comptabilisant les fréquences de chaque mot du fichier (le mot *bol* est considéré dans ce calcul);
- une table de hachage comptabilisant les fréquences de chaque paire de mots successifs dans le fichier; à ce niveau, pour chaque ligne traitée du fichier, il faut considérer le mot spécial de début de ligne (*bol*) et celui de fin de ligne (*eol*); ces mots sont à considérer dans la liste de paires de mots traités;
- le nombre de mots différents dans le fichier (variable V, vocabulaire) (sont exclues du calcul *bol* et *eol*);
- le nombre total de mots dans le fichier (variable N) (sont exclues du calcul *bol* et *eol*).

Remarque : pour les variables V et N, on ne tient pas compte des mots spéciaux de début et de fin de ligne (*bol* et *eol*) .
Exemples :

```
# let h1,h2,n1,n2 = wwpf_frequence "vide.txt";;
val h1 : (string, int) Spellc.H.t = <abstr>
val h2 : (string * string, int) Spellc.H.t = <abstr>
val n1 : int = 0
val n2 : int = 0

# H.length h1;;
- : int = 0

# H.length h2;;
- : int = 0

# let h1,h2,n1,n2 = wwpf_frequence "corpus.txt";;
val h1 : (string, int) Spellc.H.t = <abstr>
val h2 : (string * string, int) Spellc.H.t = <abstr>
val n1 : int = 17
val n2 : int = 28

# H.length h1;;
- : int = 18

# H.length h2;;
- : int = 25

# print_h id string_of_int h1;;
objective : 1
realize : 2
a : 1
back : 1
my : 1
that : 1
<s> : 5
have : 1
is : 1
the : 3
future : 2
i : 3
in : 1
to : 4
dreams : 1
hope : 2
be : 1
dream : 2
- : unit = ()
```

```
# print_h (fun (w1,w2) -> w1 ^ "," ^ w2) string_of_int h2;;
be,in : 1
hope,to : 2
objective,is : 1
future,</s> : 2
the,objective : 1
in,the : 1
that,dream : 1
i,have : 1
<s>,back : 1
realize,that : 1
i,hope : 2
is,to : 1
to,the : 1
back,to : 1
to,be : 1
dream,</s> : 2
realize,my : 1
<s>,i : 3
my,dreams : 1
dreams,</s> : 1
have,a : 1
<s>,the : 1
the,future : 2
to,realize : 2
a,dream : 1
- : unit = ()
```

3.2 Modèle de canal

Les 5 fonctions suivantes, que vous devez compléter, sont liées au jeu de données répertoriant les fautes communes dans une langue (fichier «misspell.txt» mentionné à la section 1.2.1) : lecture des données du fichier, transformation en paires de chaines de caractères (w_k,x_k), soit mots correctement orthographiés associés à mots mal orthographiés, génération des 4 matrices de confusion (voir section 1.2.1), «subm», «delm», «insm» et «tram», calcul des caractères et paires de séquences de 2 caractères successifs des mots correctement orthographies, et finalement génération d'une table de hachage, «cpcf», précisant la fréquence de ces caractères et paires de caractères successifs.

liste_words' Pour un fichier donné, ayant une syntaxe bien particulière (m_1->m_2, m_3, m_4 ...), la fonction :

- lit les lignes du fichier (on ignore les lignes vides);
- pour chaque ligne, transforme le tout en minuscule, puis appelle «split_line'» pour séparer les parties à gauche et à droite de "->"; puis, appelle «split_line"» pour identifier les différents mots séparés par ", ";
- le résultat est une liste de paires associant le mot à gauche de "->" à la liste des mots, séparés par ", ", à droite de "->".

Exemples :

```
# liste_words' "vide.txt";;
- : (string * string list) list = []

# liste_words' "misspell.txt";;
- : (string * string list) list =
[("comback", ["comeback"]); ("bcak", ["back"]); ("bakc", ["back"]);
("backrounds", ["backgrounds"]); ("backgorund", ["background"]);
("ackward", ["awkward"; "backward"]); ("relized", ["realised"; "realized"]);
("dreasm", ["dreams"]); ("deram", ["dram"; "dream"]);
("haev", ["have"; "heave"]); ("ahve", ["have"]); ("ahppen", ["happen"]);
("ahev", ["have"])]
```

word_pair_list Cette fonction :

- récupère le résultat de «liste_words'» pour un fichier donné;
- pour chaque paire (m_1,[m_2;..;m_k]), elle retourne la liste des paires [(m_2,m_1); ...; (m_k,m_1)], le tout dans une liste de paires en résultat.

Exemples :

```
# word_pair_list "vide.txt";;
- : (string * string) list = []
```

```
# word_pair_list "misspell.txt";;
- : (string * string) list =
[("comeback", "comback"); ("back", "bcak"); ("back", "bakc");
("backgrounds", "backrounds"); ("background", "backgorund");
("awkward", "ackward"); ("backward", "ackward"); ("realised", "relized");
("realized", "relized"); ("dreams", "dreasm"); ("dram", "deram");
("dream", "deram"); ("have", "haev"); ("heave", "haev"); ("have", "ahve");
("happen", "ahppen"); ("have", "ahev")]
```

gen_matrix À partir du résultat de la fonction «word_pair_list» (issu d'un fichier donné), la fonction :

- calcule la distance entre chaque mot correctement orthographié (premier élément de la paire) et le mot mal orthographié correspondant;
- pour chaque opération issue de cette distance (faute), la fonction met à jour en conséquence la matrice donnée; par exemple :
  - pour une opération DEL(c,c'), la fonction met à jour la matrice «delm», à la position (c',c);
  - pour une opération INS(c,c'), la fonction met à jour la matrice «insm», à la position (c',c);
  - pour une opération SUB(c,c'), la fonction met à jour la matrice «subm», à la position (c',c);
  - pour une opération TRANS(c,c'), la fonction met à jour la matrice «tram», à la position (c,c').

Exemples :

```
# let subm, delm, insm, tram = gen_matrix "vide.txt";;
val subm : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val delm : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val insm : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val tram : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...

# pos_diff_zero subm;;
- : (char * char * int) list = []

# pos_diff_zero delm;;
- : (char * char * int) list = []

# pos_diff_zero insm;;
- : (char * char * int) list = []

# pos_diff_zero tram;;
- : (char * char * int) list = []

# let subm, delm, insm, tram = gen_matrix "misspell.txt";;
val subm : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val delm : int array array =
[|[|0; 0; 0; 0; 0; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val insm : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
val tram : int array array =
[|[|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0;
...
```

```
# pos_diff_zero subm;;
- : (char * char * int) list = [('z', 's', 1); ('c', 'w', 1)]

# pos_diff_zero delm;;
- : (char * char * int) list =
[('v', 'e', 1); ('m', 'e', 1); ('k', 'g', 1); ('e', 'a', 2); ('#', 'b', 1)]

# pos_diff_zero insm;;
- : (char * char * int) list = [('d', 'e', 1)]

# pos_diff_zero tram;;
- : (char * char * int) list =
[('v', 'e', 2); ('r', 'o', 1); ('r', 'e', 1); ('m', 's', 1); ('h', 'a', 3);
('e', 'a', 1); ('c', 'k', 1); ('a', 'c', 1)]
```

c_cc_gen À partir d'un mot, la fonction retourne la liste des caractères et des paires de caractères successifs du mot : le résultat est une liste de chaines de caractères (*string*). Remarque :

- pour un mot vide, retourne une liste vide;
- dans la liste de caractères retournée, on tient compte du caractère spécial de début de mot (*bow*);
- dans la liste de paires de caractères successifs, on tient compte du caractère spécial de début de mot (*bow*).

Exemples :

```
# c_cc_gen "";;
- : string list = []

# c_cc_gen "a";;
- : string list = ["#"; "a"; "#a"]

# c_cc_gen "ab";;
- : string list = ["#"; "a"; "b"; "#a"; "ab"]

# c_cc_gen "abc";;
- : string list = ["#"; "a"; "b"; "c"; "#a"; "ab"; "bc"]
```

gen_table À partir du résultat de «word_pair_list», et en considérant les mots correctement orthographiés, cette fonction retourne une table de hachage associant à des caractères (sous forme de *string*) le nombre d'occurrences que l'on retrouve dans ces mots (*bol* est considérée dans ce calcul), et associant à des séquences de 2 caractères (sous forme de *string*) le nombre d'occurrences qu'on retrouve dans ces mots. Exemples :

```
# let th = gen_table "vide.txt";;
val th : (string, int) Spellc.H.t = <abstr>

# H.length th;;
- : int = 0

# let th = gen_table "misspell.txt";;
val th : (string, int) Spellc.H.t = <abstr>
```

```
# print_h id string_of_int th ;;
a : 19
#h : 5
u : 2
w : 3
ea : 5
wk : 1
# : 17
ck : 6
he : 1
p : 2
#r : 2
pe : 1
om : 1
e : 13
eb : 1
g : 2
h : 5
se : 1
pp : 1
ro : 2
r : 9
ha : 4
ze : 1
ou : 2
#d : 3
ac : 6
i : 2
re : 4
#c : 1
m : 4
is : 1
d : 9
v : 4
li : 2
b : 6
am : 3
ed : 2
kg : 2
c : 7
dr : 3
ba : 6
ve : 4
gr : 2
un : 2
wa : 2
kw : 2
nd : 2
al : 2
s : 3
n : 3
z : 1
ar : 2
#a : 1
k : 7
ap : 1
l : 2
rd : 2
aw : 1
#b : 5
av : 4
o : 3
en : 1
me : 1
ds : 1
iz : 1
ra : 1
ms : 1
co : 1
- : unit = ()
```

3.2.1 Modèle de canal

Les 2 fonctions suivantes concernent le calcul de la probabilité pour une faute donnée (opération issue du calcul d'une distance entre 2 mots), et de la probabilité pour n fautes.

p_fault À partir d'une operation (issue de «dist»), cette fonction retourne la probabilité que cette faute ait lieu (relativement à ce qui est indiqué dans l'équation 3, page 3, de l'énoncé).

Exemples :

```
# _K;;
- : int = 68

# p_fault (INS('a','b'));;
- : float = 0.013513513513513514

(* En effet, par l'eq. (3), page 3, on a: *)
# (0. +. 1.) /. (6. +. 68.);;
- : float = 0.013513513513513514

# p_fault (INS('b','a'));;
- : float = 0.011494252873563218

# p_fault (INS('e','d'));;
- : float = 0.025974025974025976

# p_fault (TRANS('h','a'));;
- : float = 0.055555555555555552

# (3. +. 1.) /. (4. +. 68.);;
- : float = 0.055555555555555552
```

prob_xw Cette fonction retourne le produit des probabilités que chacune des opérations présente dans la liste en argument ait lieu (équation 4, page 4).

Exemples :

```
# prob_xw [INS('e','d'); TRANS('h','a')];;
- : float = 0.001443001443001443

# prob_xw [INS('a','b'); INS('b','a')];;
- : float = 0.00015532774153463808
```

3.3 Modèle de langue

prob_uwv Cette fonction retourne la probabilité d'un candidat w selon les 4 cas de figure présentés dans la section 1.2.2, page 4.

Exemples :

```
# _N, _V;;
- : int * int = (28, 17)

# prob_uwv "back";;
- : float = 0.044444444444444446
# (1. +. 1.) /. (28. +. 17.);;
- : float = 0.044444444444444446

# prob_uwv "hope";;
- : float = 0.066666666666666666
# (2. +. 1.) /. (28. +. 17.);;
- : float = 0.066666666666666666

# prob_uwv "to";;
- : float = 0.11111111111111111

# prob_uwv ~u:"hope" "to";;
- : float = 0.15789473684210525
# (2. +. 1.) /. (2. +. 17.);;
- : float = 0.15789473684210525
```

```
# prob_uwv ~u:"realize" "to";;
- : float = 0.052631578947368418
# (0. +. 1.) /. (2. +. 17.);;
- : float = 0.052631578947368418

# prob_uwv ~u:"i" "hope";;
- : float = 0.15

# prob_uwv ~u:"<s>" "hope";;
- : float = 0.045454545454545456
# (0. +. 1.) /. (5. +. 17.);;
- : float = 0.045454545454545456

# prob_uwv ~u:"<s>" "i";;
- : float = 0.18181818181818182

# prob_uwv "future" ~v:"</s>";;
- : float = 0.010526315789473684

# prob_uwv "dream" ~v:"</s>";;
- : float = 0.010526315789473684
# ((2. +. 1.) /. (28. +. 17.)) *. ((2. +. 1.) /. (2. +. 17.));;
- : float = 0.010526315789473684

# prob_uwv "hope" ~v:"to";;
- : float = 0.010526315789473684

# prob_uwv "to" ~v:"be";;
- : float = 0.010582010582010581

# prob_uwv ~u:"i" "hope" ~v:"to";;
- : float = 0.023684210526315787

# prob_uwv ~u:"the" "future" ~v:"</s>";;
- : float = 0.023684210526315787

# prob_uwv ~u:"to" "realize" ~v:"my";;
- : float = 0.015037593984962405

# prob_uwv ~u:"to" "realize" ~v:"that";;
- : float = 0.015037593984962405

# prob_uwv ~u:"the" "future" ~v:"</s>";;
- : float = 0.023684210526315787

# prob_uwv ~u:"<s>" "i" ~v:"have";;
- : float = 0.018181818181818184

# prob_uwv ~u:"<s>" "i" ~v:"hope";;
- : float = 0.027272727272727271
```

3.4 Étapes de l'algorithme présenté à la page 2 de l'énoncé

Les 2 fonctions suivantes correspondent aux étapes 2 et 3 précisées à la section 1.2, page 2, de l'énoncé.

find_candidates Cette fonction retourne la liste des candidats *w*, de la table «wf», tel que leur taille est égale, plus ou moins 2, à celle du mot *x* erroné passé en argument, et tel que leur distance par rapport à *x* est inférieure ou égale à *k*. le résultat est une liste associant à chaque candidat *w* identifié la liste des opérations requises pour passer de ce mot au mot erroné *x*.

Exemples :

```
# find_candidates "hve";;
- : (string * Spellc.operation list) list =
[("be", [SUB ('b', 'h'); INS ('v', 'b')]);
("hope", [SUB ('o', 'v'); DEL ('p', 'o')]);
("the", [DEL ('t', '#'); INS ('v', 'h')]); ("have", [DEL ('a', 'h')])]

# find_candidates "dram";;
- : (string * Spellc.operation list) list =
[("dream", [DEL ('e', 'r')]); ("dreams", [DEL ('e', 'r'); DEL ('s', 'm')])]
```

```
# find_candidates "futuer";;
- : (string * Spellc.operation list) list = [("future", [TRANS ('r', 'e')])]

# find_candidates "te";;
- : (string * Spellc.operation list) list =
[("be", [SUB ('b', 't')]); ("to", [SUB ('o', 'e')]);
("in", [SUB ('i', 't'); SUB ('n', 'e')]);
("i", [SUB ('i', 't'); INS ('e', 'i')]); ("the", [DEL ('h', 't')]);
("is", [SUB ('i', 't'); SUB ('s', 'e')]);
("my", [SUB ('m', 't'); SUB ('y', 'e')]);
("a", [SUB ('a', 't'); INS ('e', 'a')])]

# find_candidates "bck";;
- : (string * Spellc.operation list) list =
[("be", [SUB ('e', 'c'); INS ('k', 'e')]); ("back", [DEL ('a', 'b')])]

# find_candidates "ope";;
- : (string * Spellc.operation list) list =
[("be", [SUB ('b', 'o'); INS ('p', 'b')]); ("hope", [DEL ('h', '#')]);
("the", [SUB ('t', 'o'); SUB ('h', 'p')])]
```

best_candidate À partir d'un mot erroné et une liste de candidat, cette fonction retourne le meilleur candidat en considérant le modèle de canal et le modèle de langue (voir énoncé; la fonction est fournie et correspond, dans le code, à «prob_xw_uwv»).

Remarque : Si la liste des candidats est vide, la fonction retourne le mot erroné.

Exemples :

```
# best_candidate "hve" (find_candidates "hve");;
- : string = "have"

# best_candidate "dram" (find_candidates "dram");;
- : string = "dream"

# best_candidate "futuer" (find_candidates "futuer");;
- : string = "future"

# best_candidate "te" (find_candidates "te");;
- : string = "to"

# best_candidate "bck" (find_candidates "bck");;
- : string = "back"

# best_candidate "ope" (find_candidates "ope");;
- : string = "hope"

# best_candidate ~u:"i" "hve" (find_candidates "hve");;
- : string = "have"

# best_candidate ~u:"i" "hve" ~v:"to" (find_candidates "hve");;
- : string = "have"

# best_candidate ~u:"in" "te" (find_candidates "te");;
- : string = "the"

# best_candidate ~u:"in" "te" ~v:"future" (find_candidates "te");;
- : string = "the"
```

3.5 Fonctions principales

À part la fonction «tri_gramme», les 2 fonctions qui suivent correspondent aux 2 principales fonctions du travail pratique : la première pour la correction automatique d'un texte, et la deuxième pour la *completion* automatique de textes saisis, en tenant compte du contexte et en corrigeant si nécessaire le texte saisi.

tri_gramme À partir d'une liste de mots, cette fonction retourne une liste de triplets (tri-gramme) de mots successifs; elle tient compte du début et de la fin de phrase.

Exemples :

```
# tri_gramme [];;
- : (string * string * string) list = []

# tri_gramme ["m"];;
- : (string * string * string) list = [("<s>", "m", "</s>")]

# tri_gramme ["m1"; "m2"];;
- : (string * string * string) list = [("<s>", "m1", "m2"); ("m1", "m2", "</s>")]

# tri_gramme ["m1"; "m2"; "m3"];;
- : (string * string * string) list =
[("<s>", "m1", "m2"); ("m1", "m2", "m3"); ("m2", "m3", "</s>")]
```

revise_lwords Principale fonction de correction automatique : elle prend une liste de mots, formant une phrase, et retourne la liste des mots éventuellement corrigés. La fonction prend naturellement avantage des tri-grammes; elle est appelée par «revise_line» qui prend une chaine de caractères, la découpe en mots, en en retenant les séparateurs, les majuscules, etc., puis invoque «revise_lwords» pour corriger les mots, avant de remettre le texte comme il était (séparateurs, etc.)

Exemples :

```
# (revise_lwords % split_line) "i␣hve␣a␣dream";;
- : string list = ["i"; "have"; "a"; "dream"]

# (revise_lwords % split_line) "i␣hpe␣to␣be";;
- : string list = ["i"; "have"; "to"; "be"]

# (revise_lwords % split_line) "i␣hpe␣to␣be␣in␣the␣future";;
- : string list = ["i"; "hope"; "to"; "understand"; "the"; "meaning"]

# (revise_lwords % split_line) "to␣reaize␣my␣fuure";;
- : string list = ["to"; "realize"; "my"; "future"]
```

find_cand_aux Deuxième principale fonction du travail : elle prend en argument un mot (préfixe saisi par l'utilisateur), qui peut être vide, et retourne une liste de candidats triés selon le plus probable qui devrait apparaitre dans le contexte (notant qu'on ne dispose que du mot précédent (qui peut être *bol*)); ainsi, cette fonction est typiquement invoquée lorsqu'on saisit en temps réel un texte et qu'on veut que le système nous propose, à chaque fois, une liste de mots candidats à compléter pour ce qui est saisi (les mots proposés correspondent éventuellement à une correction du «mot» ou «préfixe» saisi par l'utilisateur).

L'algorithme fonctionne comme suit :

- identification des candidats potentiels et de leur probabilité :
  - si le mot est vide (l'utilisateur n'a encore rien saisi), la fonction retourne la liste des mots de «wpf» tel que le prédécesseur correspond à l'argument *u* (qui peut être *bol*, ou le mot saisi précédemment par l'utilisateur); pour chaque mot retourné, est associé la probabilité que ce mot apparaisse dans le contexte (c'est-à-dire en considérant le mot précédent u);
  - sinon, elle tente de calculer la liste des mots ayant le préfixe saisi (Tp1); et une nouvelle fois, pour chaque mot retourné, est associé la probabilité que ce mot apparaisse dans le contexte (c'est-à-dire en considérant le mot précédent u);
  - si elle ne réussit pas (exception Not_found soulevée par la fonction liste_par_prefixe du module «Dico»), à partir de «wpf», elle identifie les mots tel qu'ils soient de la taille du mot en argument, plus ou moins 2, et telle que leur distance par rapport au mot/préfixe saisi est inférieure ou égale à *k* (par défaut, égale à 2); à ce niveau, la probabilité associée à chaque mot correspond à celle qui tient compte du contexte, mais aussi de la probabilité des fautes (ainsi, il s'agit d'appeler la fonction «prob_xw_uwv»);
- une fois la liste des candidats identifiés, on les trie selon la probabilité qui leur est associée.
- on retourne alors les candidats triés (chacun accompagné de sa probabilité).

Exemples :

```
# find_cand_aux "";;
- : (string * float) list =
[("i", 0.18181818181818182); ("the", 0.090909090909090912); ("back", 0.090909090909090912)]
```

```
# find_cand_aux "i";;
- : (string * float) list =
[("i", 0.18181818181818182); ("is", 0.045454545454545456); ("in", 0.045454545454545456)]

# find_cand_aux "bck";;
- : (string * float) list = [("back", 0.0012285012285012287); ("be", 6.9279904670851165e-006)]

# find_cand_aux "hop";;
- : (string * float) list = [("hope", 0.045454545454545456)]

# find_cand_aux "t";;
- : (string * float) list =
[("the", 0.090909090909090912); ("to", 0.045454545454545456); ("that", 0.045454545454545456)]

# find_cand_aux ~u:"is" "t";;
- : (string * float) list =
[("to", 0.11111111111111111); ("the", 0.055555555555555552); ("that", 0.055555555555555552)]

# find_cand_aux ~u:"in" "t";;
- : (string * float) list =
[("the", 0.11111111111111111); ("to", 0.055555555555555552); ("that", 0.055555555555555552)]

# find_cand_aux "dram";;
- :(string * float) list = [("dream", 0.00063131313131313126); ("dreams", 9.1494656712048015e-006)]

# find_cand_aux "drem";;
- : (string * float) list = [("dream", 0.00186799501867995); ("dreams", 2.7072391575071739e-005)]

# find_cand_aux  ~u:"my" "dram";;
- : (string * float) list = [("dream", 0.0007716049382716049); ("dreams", 2.2365360529611736e-005)]

# find_cand_aux  ~u:"a" "dram";;
- : (string * float) list = [("dream", 0.0015432098765432098); ("dreams", 1.1182680264805868e-005)]
```

4 Exploitation du TP

Le fichier «gui2.ml» comprend la définition d'un module permettant d'exploiter concrètement la notion de *complétion* de mots, en tenant compte du contexte et des mots mal orthographiés, lors de la saisie d'un texte (le tout dans une interface graphique assez simple). Pour utiliser ce module, une fois que votre Tp2 complété, il faut simplement charger le fichier :

```
# #use "gui2.ml";;
(* Charge le fichier et affiche une interface graphique permettant la saisie
   de texte et offrant la completion et la correction de ce qui est saisi
*)
```

5 Démarche à suivre

Tel que le fichier «dico.ml» a été conçu (utilisation d'exceptions pour les fonctions non implantées), vous pouvez charger le fichier et exploiter quelques fonctions déjà implantées :

```
# #use "spellc.ml";;
module Dico :
...sig
module Utiles :
...sig
module Spellc :
...

# open Utiles;;
# open Spellc;;

# dist "youtube" "yotube";;
- : int * Spellc.operation list = (1, [DEL ('u', 'o')])

# dist "youtube" "youttube";;
- : int * Spellc.operation list = (1, [INS ('t', 'u')])
```

Ne vous reste plus qu'à compléter une à une les fonctions à implanter et à les tester progressivement.

6 À remettre

Il faut remettre un fichier .zip contenant uniquement le fichier «spellc.ml» complété. Le code doit être clair et bien indenté[4].

7 Remarques importantes

Prix Pierre Ardouin : Ce travail est éligible à ce Prix. Parmi les contributions possibles :

- l'ajout dans un fichier «contrib.ml» (qui utilise le fichier "spellc.ml") de fonctions permettant d'enrichir les fonctionnalités du travail pratique;
- l'amélioration de l'interface («gui2.ml») :
  - utilisation d'autres paquetages disponibles dans l'écosystème d'OCaml : lablgtk2 ou labltk;
  - utilisation d'une interface Web;
- autres contributions.

Plagiat : Tel que décrit dans le plan de cours, le plagiat est interdit. Une politique stricte de tolérance zéro est appliquée en tout temps et sous toutes circonstances. Tous les cas seront référés à la direction de la Faculté.

Travail remis en retard : Tout travail remis en retard se verra pénalisé de 25% par jour de retard. Chaque journée de retard débute dès la limite de remise dépassée (dès la première minute). Un retard excédant 2 jours (48h) provoquera le rejet du travail pour la correction et la note de 0 pour ce travail. La remise doit se faire par la boite de dépôt du TP2 dans la section «Évaluation et résultats».

Barème : Au niveau des fonctions à implanter, le barème est précisé dans le fichier «spellc.ml». Notez cependant que :

- (-10pts), si votre code ne compile pas (provoque une erreur/exception lors du chargement du fichier «spellc.ml» dans l'interpréteur, i.e. «#**use** "spellc.ml";;»);
- (-5pts), si votre code n'est pas clair et bien indenté.

Bon travail.

[4] Suggestion : http://ocaml.org/learn/tutorials/guidelines.html