2 Constitution d'un lexique (en C)

Question 1

On propose la fonction suivante :

```c
int lexcmp(char* s1, char* s2){
    int i = 0;
    while (s1[i] != '\0' && s2[i] != '\0'){
        if (s1[i] < s2[i]){
            return -1;
        } else if (s1[i] > s2[i]) {
            return 1;
        }
        i++;
    }
    if (s1[i] != '\0'){
        return 1;
    } else if (s2[i] != '\0') {
        return -1;
    }
    return 0;
}
```

Question 2

Tant qu'on n'est pas arrivé à la fin de l'un des deux mots, on compare les lettres une à une. On sort de la boucle si l'un des deux mots est préfixe de l'autre. On vérifie alors pour quel mot on a atteint la fin.

La complexité spatiale est constante (on ne crée que des variables entières). La complexité temporelle est linéaire en $\min(|s_1|, |s_2|)$, car le corps de la boucle se fait en temps constant, et on arrête la boucle lorsqu'on atteint la fin du mot le plus court.

Question 3

La fonction `delete_min` supprime l'élément minimal d'un arbre binaire de recherche et renvoie l'arbre résultant. La valeur de retour est nécessaire, notamment lorsque l'élément minimal se trouve à la racine. De plus, la fonction prend en argument un pointeur vers un pointeur de nœud, et modifie la valeur référencée en y stockant le nœud qui contient le minimum. Cela permet notamment d'éviter une fuite mémoire, car on dispose d'un pointeur vers le nœud supprimé, et permet de réutiliser ce nœud dans la fonction suivante.

La fonction `delete` prend en argument une clé et un arbre binaire de recherche et supprime le nœud qui contient la clé. Si la racine n'est pas la clé cherchée, la fonction continue récursivement à gauche ou à droite selon la comparaison, sinon, on remplace la racine par le plus petit nœud du fils droit, qu'on aura extrait avec la fonction précédente. La fonction libère ensuite la clé du nœud à supprimer (ce qui sous-entend qu'elle est allouée sur le tas), ainsi que le nœud à supprimer.

Question 4

On propose deux fonctions classiques sur les arbres binaires de recherche, ainsi qu'une fonction de libération mémoire :

- une fonction libérant la mémoire utilisée par un arbre binaire de recherche. On détruit récursivement les fils gauche et droit avant de libérer la mémoire occupée par le nœud.

```
void bst_destroy(node* bst){
    if (bst != NULL){
        bst_destroy(bst->left);
        bst_destroy(bst->right);
        free(bst->key);
        free(bst);
    }
}
```

– une fonction de recherche, qui renvoie le nombre d'occurrences associé à un mot (on compare la clé avec la racine et on continue à gauche ou à droite si on n'a pas trouvé la bonne clé). Cette fonction est un classique des arbres binaires de recherche, mais il s'avère qu'elle ne sera pas utilisée par la suite. On donne quand même le code :

```
int get(char* key, node* bst){
    if (bst != NULL){
        int c = lexcmp(key, bst->key);
        if (c == 0){
            return bst->val;
        } else if (c < 0) {
            return get(key, bst->left);
        }
        return get(key, bst->right);
    }
    return 0;
}
```

– une fonction d'insertion qui ajoute un nombre d'occurrences associé à un mot, en créant un nouveau nœud si nécessaire (et renvoie le nouvel arbre ainsi formé). En cohérence avec la remarque de la question précédente, on alloue la clé sur le tas le cas échéant.

```
node* insert(char* key, int val, node* bst){
    if (bst == NULL){
        bst = malloc(sizeof(node));
        int len = strlen(key);
        bst->key = malloc((len + 1) * sizeof(char));
        strcpy(bst->key, key);
        bst->val = val;
        bst->left = NULL;
        bst->right = NULL;
    } else {
        int c = lexcmp(key, bst->key);
        if (c == 0){
            bst->val += val;
        } else if (c < 0) {
            bst->left = insert(key, val, bst->left);
        } else {
            bst->right = insert(key, val, bst->right);
        }
    }
    return bst;
}
```

Question 5

Voir les commentaires précédents.

Question 6

Il faut pouvoir tester différents cas de figure, à savoir recherche et insertion dans :

- un arbre vide ;
- un arbre non vide ne contenant pas la clé ;
- un arbre contenant la clé.

On propose, par exemple :

```
node* bst = NULL;
char foo[4] = "foo",
     bar[4] = "bar",
     baz[4] = "baz",
     qux[4] = "qux",
     quux[5] = "quux";
printf("Recherche dans un arbre vide : %d\n", get(foo, bst));
bst = insert(foo, 3, bst);
bst = insert(bar, 4, bst);
bst = insert(baz, 5, bst);
bst = insert(qux, 1, bst);
printf("Recherche dans un arbre ne contenant pas le nœud : %d\n", get(quux, bst));
printf("Recherche dans un arbre contenant le nœud : %d\n", get(qux, bst));
bst = insert(qux, 2, bst);
printf("Insertion dans un arbre contenant le nœud : %d\n", get(qux, bst));
bst_destroy(bst);
```

Question 7

On propose de découper le code en plusieurs morceaux :

- une fonction pour construire un arbre binaire de recherche à partir d'un fichier texte. Tant qu'on n'est pas arrivé à la fin du fichier, on lit chaque ligne et on insère le mot dans l'arbre.

```
node* build_bst(char* pathname){
    FILE* file = fopen(pathname, "r");
    node* bst = NULL;
    char line[101];
    int success = fscanf(file, "%s", line);
    while (success != EOF){
        bst = insert(line, 1, bst);
        success = fscanf(file, "%s", line);
    }
    fclose(file);
    return bst;
}
```

- une fonction qui écrit le parcours en profondeur infixe d'un arbre binaire de recherche dans un fichier ouvert en mode écriture.

```
void bst_to_lex(node* bst, FILE* file){
    if (bst != NULL){
        bst_to_lex(bst->left, file);
        fprintf(file, "%s %d\n", bst->key, bst->val);
        bst_to_lex(bst->right, file);
    }
}
```

- une fonction qui prend en argument des noms de fichiers d'entrée et de sortie et construit le lexique associé à un fichier texte. La fonction renvoie l'arbre construit.

```c
node* text_to_lex(char* input, char* output){
    node* bst = build_bst(input);
    FILE* file = fopen(output, "w");
    bst_to_lex(bst, file);
    fclose(file);
    return bst;
}
```

Question 8

On trouve les informations suivantes :

	Les trois mousquetaires	La comédie humaine
Mots différents	14677	61551
Mot le plus fréquent et nombre d'occurrences	, (virgule) : 21499 fois	, (virgule) : 214697 fois
Nombre de mots apparaissant 10 fois	146	824
Hauteur de l'arbre binaire	35	38

On a utilisé les fonctions suivantes pour faire ces calculs :

– calcul de taille :

```c
int size(node* bst){
    if (bst == NULL){
        return 0;
    }
    return 1 + size(bst->left) + size(bst->right);
}
```

– garde en mémoire le nœud avec le plus grand nombre d'occurrences :

```c
void most_frequent(node* bst, node** best){
    if (bst != NULL){
        if (*best == NULL || bst->val > (*best)->val){
            *best = bst;
        }
        most_frequent(bst->left, best);
        most_frequent(bst->right, best);
    }
}
```

– compte le nombre de mots ayant un certain nombre d'occurrences :

```c
void number_exact_occurrences(node* bst, int occ, int* nb_words){
    if (bst != NULL){
        if (bst->val == occ){
            (*nb_words)++;
        }
        number_exact_occurrences(bst->left, occ, nb_words);
        number_exact_occurrences(bst->right, occ, nb_words);
    }
}
```

– calcul de hauteur :

```c
int height(node* bst){
    if (bst == NULL){
        return -1;
    }
    int hleft = height(bst->left);
    int hright = height(bst->right);
    return 1 + ((hleft > hright)?hleft:hright);
}
```

Question 9

Pour un texte de taille N, il faut faire de l'ordre de N insertions, ce qui peut se faire en temps $\Theta(N^2)$ dans le pire cas, car l'arbre binaire de recherche ainsi obtenu peut être de hauteur linéaire en N (par exemple si les mots apparaissent dans l'ordre alphabétique pour la première fois). On pourrait utiliser des arbres auto-équilibrés, comme les arbres Rouge-Noir, pour garantir une complexité en $\Theta(N \log N)$ dans le pire cas.

On constate toutefois à la question précédente que les arbres semblent équilibrés (la hauteur est environ 15 fois $\log_2 N$ dans les deux cas). Cela peut s'expliquer par le fait que les mots apparaissent pour la première fois dans un ordre d'apparence aléatoire, ce qui évite un déséquilibre de l'arbre.

Question 10

C'est exactement la même idée que la construction du lexique, sauf qu'on considère un bigramme comme une seule chaîne de caractères constituée des deux mots séparés par une espace. Il y a quelques précautions à prendre lors du parcours du fichier, mais c'est tout.

```c
node* build_bst_big(char* pathname){
    FILE* file = fopen(pathname, "r");
    node* bst = NULL;
    char line[101];
    char big[202];
    int success = fscanf(file, "%s", line);
    while (success != EOF){
        strcpy(big, line);
        int len = strlen(line);
        success = fscanf(file, "%s", line);
        big[len] = ' ';
        strcpy(&big[len + 1], line);
        bst = insert(big, 1, bst);
    }
    fclose(file);
    return bst;
}

void text_to_big(char* input, char* output){
    node* bst = build_bst_big(input);
    FILE* file = fopen(output, "w");
    bst_to_lex(bst, file);
    fclose(file);
    bst_destroy(bst);
}
```

3 Segmentation d'un texte (en OCaml)

Question 11

On adapte le code fourni pour ajouter les éléments au dictionnaire. On pense à convertir le nombre d'occurrences en entier.

```
let lire_lexique nom_fichier =
    let h = Hashtbl.create 4 in
    let f = open_in nom_fichier in
    begin
        try while true do
            match String.split_on_char ' ' (input_line f) with
                | [mot; occ] -> Hashtbl.add h mot (int_of_string occ)
                | _ -> failwith "Erreur de format"
        done
        with End_of_file -> close_in f
    end;
    h
```

Question 12

On trouve les réponses suivantes :

- il y a 31337 mots différents, en utilisant `Hashtbl.length` ;
- le mot le plus long est de longueur 78, on l'a trouvé avec le code suivant :

```
let plus_long_mot h =
    (* Détermine le mot le plus long dans un lexique. *)
    let comparer mot occ mot_max =
        if String.length mot > String.length mot_max then mot
        else mot_max in
    Hashtbl.fold comparer h ""
```

- le nombre de mots n'apparaissant qu'une seule fois est 14150, on l'a trouvé avec le code :

```
let nb_mots_uniques h =
    let nb = ref 0 in
    Hashtbl.iter (fun mot occ -> if occ = 1 then incr nb) h;
    !nb
```

Question 13 Pas de difficulté ici :

```
let est_mot h mot = Hashtbl.mem h mot

let score h mot =
    if est_mot h mot then log (float_of_int (Hashtbl.find h mot))
                     else neg_infinity
```

Question 14

L'approche gloutonne la plus directe serait le lire le texte et de faire un découpage chaque fois qu'on trouve un mot qui existe. Malheureusement, cette approche peut ne pas marcher. Par exemple, pour la chaîne `beaucoupdevin`, un découpage glouton donnerait `beau cou pdevin`, ce qui n'est pas une segmentation valide, car `pdevin` n'est pas un mot.

Question 15

Un arbre préfixe (ou *trie*) permettrait de résoudre le problème efficacement : à chaque nouvelle lettre lue, on continue sur le fils correspondant à cette lettre. Si un tel fils n'existe pas, il n'existe pas de segmentation. Lorsqu'on atteint une feuille, on crée le mot correspondant et on recommence à la racine.

Question 16

L'algorithme de Huffman part du principe que pour compresser un texte, il vaut mieux encoder les lettres les plus fréquentes par des petits mots de code. Il est optimal (en espace) pour la compression sans perte et construit un arbre permettant la compression et la décompression :

- la construction de l'arbre de Huffman est un algorithme glouton :
  - créer une file de priorité des couples $(\text{freq}(a), \texttt{Feuille}(a))$ pour chaque lettre a ;
  - tant qu'il reste au moins deux arbres, extraire les deux de plus petite fréquence (f_1, A_1) et (f_2, A_2) et insérer $(f_1 + f_2, \texttt{Nœud}(A_1, A_2))$;
  - l'arbre final est l'arbre de Huffman ;
- la complexité de la construction est en $\mathcal{O}(n \log n)$, où n est la taille du texte à compresser. Le facteur log vient de l'utilisation d'une file de priorité ;
- l'encodage d'une lettre se fait en suivant le chemin de la racine jusqu'à la feuille qui contient cette lettre : un 0 pour une branche gauche, un 1 pour une branche droite ;
- la décompression se fait en lisant le texte compressé : si on lit un 0 (resp. 1), on emprunte la branche de gauche (resp. droite), si on arrive sur une feuille, on écrit la lettre et on remonte à la racine.

Question 17

On propose de représenter une segmentation d'un mot u comme une liste de tailles $j_1, j_2, \ldots, j_k$, toutes > 0, correspondant aux tailles des mots de la segmentation.

La recherche des préfixes consiste juste à parcourir $i \in [\![1, |u|]\!]$ et à tester si $u[0 : i]$ est un mot du lexique. La construction de $u[0 : i]$ prendra un temps $\mathcal{O}(i)$, ce qui donnera une complexité quadratique en $|u|$. La fonction finale aura toutefois une complexité exponentielle dans le pire cas.

Question 18 Pour tester l'existence d'une segmentation, on utilise une fonction récursive, qui va tester, si nécessaire, toutes les tailles du premier mot, puis faire un appel récursif sur le texte restant. Ici, la fonction récursive teste s'il existe une segmentation de $t[i : n]$ dont le premier mot est $t[i : i + k]$, avec $k \geqslant j$.

```
let existe_segmentation t h =
    let n = String.length t in
    let rec existe_rec i j =
        i = n ||
        (i + j <= n && est_mot h (String.sub t i j) && existe_rec (i + j) 1) ||
        (i + j < n && existe_rec i (j + 1)) in
    existe_rec 0 1
```

On adapte le code pour trouver une segmentation en construisant la liste au fur et à mesure. La fonction auxiliaire renvoie un couple (booléen, segmentation de $t[i : n]$) dont le premier mot est un $t[i : i + k]$ avec $k \geqslant j$, s'il en existe une. Le booléen vaut `false` et la liste est vide s'il n'en existe pas.

```
let trouver_segmentation t h =
    let n = String.length t in
    let rec trouver_rec i j =
        if i = n then true, []
        else if i + j > n then false, []
        else if est_mot h (String.sub t i j) then
                let b, lst = trouver_rec (i + j) 1 in
                if b then true, j :: lst
                else trouver_rec i (j + 1)
            else trouver_rec i (j + 1) in
    snd (trouver_rec 0 1)
```

Pour trouver le texte segmenté associé, on peut utiliser la fonction suivante qui reconstruit le texte à partir de la liste des tailles de mots :

```
let liste_vers_mot t seg =
    let mot = ref "" in
    let rec completer i = function
        | []     -> !mot
        | [j]    -> mot := !mot ^ String.sub t i j; !mot
        | j :: q -> mot := !mot ^ String.sub t i j ^ " ";
                    completer (i + j) q in
    completer 0 seg
```

Question 19 Pour dénombrer les segmentations, on propose le code suivant, en utilisant une boucle cette fois-ci, plutôt qu'un deuxième argument à la fonction récursive. Cette dernière compte le nombre de segmentations de $t[i:n]$ en envisageant toutes les tailles possibles de premier mot.

```
let compter_segmentation t h =
    let n = String.length t in
    let rec compte_rec i =
        if i = n then 1
        else begin
            let nb = ref 0 in
            for j = 1 to n - i do
                if est_mot h (String.sub t i j) then
                    nb := !nb + compte_rec (i + j)
            done;
            !nb
        end in
    compte_rec 0
```

Question 20 On obtient 3588 segmentations différentes du texte proposé.

Question 21 On remarque que le pire cas pour la fonction précédente est lorsque tous les facteurs du texte sont des mots du lexique. Dès lors, la complexité de la fonction auxiliaire vérifiera :

$$C(i) = \begin{cases} \Theta(1) & \text{si } i = n \\ \sum_{j=1}^{n-i} (C(i+j) + \Theta(j)) & \text{sinon} \end{cases}$$

On remarque alors que si $i < n$, alors $C(i) = 2C(i+1) + \Theta(n-i)$. Cette formule se résout en $C(0) = \Theta(2^n)$, ce qui est la complexité cherchée.

Question 22

Dans le texte `beaucoupdevin`, on calculera le nombre de segmentations de `devin` deux fois : une fois lorsqu'on aura envisagé le découpage `beaucoup` et une autre fois pour `beau coup`. Pour un texte plus long, un très grand nombre de calculs sont faits plusieurs fois. Pour corriger ce problème, on peut utiliser de la **programmation dynamique**. Ici, en récursif, cela consiste simplement à **mémoïser** les résultats déjà calculés, dans un tableau par exemple. On met initialement des valeurs -1 pour tester si on a déjà calculé une valeur ou non. Lors d'un appel récursif, on commence par tester si on a déjà calculé la valeur associée, pour ne pas la recalculer le cas échéant.

```
let compter_segmentation_memo t h =
    let n = String.length t in
    let tab = Array.make (n + 1) (-1) in
    tab.(n) <- 1;
    let rec compte_memo i =
        if tab.(i) = -1 then begin
            tab.(i) <- 0;
            for j = 1 to n - i do
                if est_mot h (String.sub t i j) then
                    tab.(i) <- tab.(i) + compte_memo (i + j)
            done;
        end;
        tab.(i) in
    compte_memo 0
```

Question 23 On trouve alors 3 803 264 741 980 241 920 segmentations du texte de préface.

Question 24 On calcule ici un tableau contenant des couples (score maximal, segmentation) à partir d'un certain indice

```
let segmentation_score_max t h =
    let n = String.length t in
    let tab = Array.make (n + 1) (-1., []) in
    tab.(n) <- (0., []);
    let rec seg_score_rec i =
        if fst tab.(i) < 0. then begin
            tab.(i) <- 0., [];
            for j = 1 to n - i do
                let u = String.sub t i j in
                let sc, seg = seg_score_rec (i + j) in
                let nouv_sc = sc +. score h u in
                if nouv_sc > fst tab.(i) then
                    tab.(i) <- nouv_sc, j :: seg
            done
        end;
        tab.(i) in
    seg_score_rec 0
```

Question 25 On obtient dès lors la segmentation suivante : `la que l le est la me il le u r`. Pour le texte de préface, le score maximal est $502,9$ environ.

Question 26 C'est quasiment le même code que la fonction précédente, en rajoutant la taille au carré du mot.

```
let segmentation_score_max_bis t h =
    let n = String.length t in
    let tab = Array.make (n + 1) (-1., []) in
    tab.(n) <- (0., []);
    let rec seg_score_rec i =
        if fst tab.(i) < 0. then begin
            tab.(i) <- 0., [];
            for j = 1 to n - i do
                let u = String.sub t i j in
                let sc, seg = seg_score_rec (i + j) in
                let nouv_sc = sc +. score h u +. float_of_int (j * j) in
                if nouv_sc > fst tab.(i) then
                    tab.(i) <- nouv_sc, j :: seg
            done
        end;
        tab.(i) in
    seg_score_rec 0
```

Question 27 On obtient ici des segmentations beaucoup plus convaincantes, mais il y a un problème pour interpréter le morceau `en os et en is` correctement, car `os` et `is` ne correspondent pas ici à des mots, mais à des suffixes de noms (Athos, Porthos et Aramis).

Question 28 On adapte le code en reconstruisant les bigrammes et en calculant le score dans le dictionnaire associé. À nouveau, presque aucune modification.

```
let segmentation_score_max_big t h hbig =
    let n = String.length t in
    let tab = Array.make (n + 1) (-1., []) in
    tab.(n) <- (0., []);
    let rec seg_score_rec i =
        if fst tab.(i) < 0. then begin
            tab.(i) <- 0., [];
            for j = 1 to n - i do
                let u = String.sub t i j in
                let sc, seg = seg_score_rec (i + j) in
                let sc_big = if i + j = n || seg = [] then 0.
                    else score hbig (u ^ " " ^ String.sub t (i + j) (List.hd seg)) in
                let nouv_sc = sc +. score h u +. float_of_int (j * j) +. sc_big in
                if nouv_sc > fst tab.(i) then
                    tab.(i) <- nouv_sc, j :: seg
            done
        end;
        tab.(i) in
    seg_score_rec 0
```

Pour construire le dictionnaire de bigrammes, on adapte la fonction `lire_lexique` :

```
let lire_big nom_fichier =
    let h = Hashtbl.create 4 in
    let f = open_in nom_fichier in
    begin
        try while true do
            match String.split_on_char ' ' (input_line f) with
                | [mot1; mot2; occ] -> Hashtbl.add h (mot1 ^ " " ^ mot2)
                                                     (int_of_string occ)
                | _ -> failwith "Erreur de format"
        done
        with End_of_file -> close_in f
    end;
    h
```

Question 29 En reprenant les trois textes de la question 27, on constate deux choses :

- le texte est segmenté parfaitement en utilisant les dictionnaires pour le lexique et les bigrammes des trois mousquetaires ;
- aucun texte n'est trouvé pour les deux autres lexiques. Ce point s'explique notamment par le constat de la question 27 (le mot `is` n'existant pas en français, il n'y a pas de bigramme associé).

Question 30 Une approche possible est de mettre un score nul à des mots absents du lexique, au lieu d'un score $-\infty$. De cette manière, on n'éliminera pas complètement une segmentation peu convaincante.