Projet : calcul de statistiques sur les mots d'un texte
Semestre pair 2023-2024

1 Introduction

Le but de ce projet est de calculer des statistiques élémentaires sur un fichier texte (longueur moyenne d'un mot, d'une phrase, fréquence d'apparition d'un mot...) en mettant en œuvre les principes vus en CTD et TP.

1.1 Intégrité académique

Le travail doit être réalisé individuellement. Vous pouvez discuter du sujet avec vos camarades mais il est strictement interdit de copier-coller du code. L'équipe pédagogique utilise des outils de mesure de similarité de codes sources. Tout plagiat sera sévèrement sanctionné.

1.2 Évaluation

Le travail que vous allez produire sera évalué :

- à partir d'un dépôt sur Moodle que vous devrez effectuer au plus tard le vendredi 10 mai 2024 à 23h59 (n'attendez pas le dernier moment...) ; quelle que soit la raison, si le dépôt est vide le 10 mai à minuit alors vous aurez 0 ; il est donc conseillé de déposer votre projet avant, même s'il n'est pas fini ; mais la version déposée doit pouvoir être compilée sans erreur ;
- à partir d'un oral qui se déroulera la semaine du 13 mai (cf. emploi du temps pour le créneau précis de votre groupe).

1.3 Barème

Le projet comporte trois niveaux de difficulté croissante, numérotés de 1 à 3. La réalisation complète et exacte de chaque niveau donne la possibilité d'obtenir une certaine note maximale, comme cela est indiqué dans le tableau suivant :

Niveau	Note maximale
1	12
2	16
3	20

Pour commencer un niveau, il faut avoir complètement réalisé le niveau précédent. Par exemple, si vous n'avez pas terminé le niveau 2 et si vous réalisez une partie du niveau 3, vous serez tout de même noté sur 16. Un niveau est considéré comme réalisé quand toutes ses fonctionnalités sont programmées et que le programme correspondant au niveau s'exécute sans erreur.
Attention : si le projet que vous avez déposé sur Moodle ne peut pas être compilé sans erreur en utilisant la commande `make`, vous aurez automatiquement la note 0.

1.4 Dépôt sur Moodle

Vous déposerez sur Moodle un seul fichier archive au format ZIP qui doit être produit en suivant les directives suivantes.

1. Créez un répertoire dont le nom doit suivre le modèle :
   NOM-PRENOM-NUMETU
   où :
   - NOM, PRENOM et NUMETU doivent être remplacés, respectivement et dans cet ordre, par votre nom de famille en majuscules, votre prénom en majuscules et votre numéro d'étudiant (8 chiffres) ;
   - n'apparaît aucune lettre accentuée ;
   - n'apparaît aucun espace ni apostrophe, les éventuels espaces et apostrophes dans votre nom ou votre prénom devant être remplacés par des tirets - (signe moins).
2. Placez dans ce répertoire :
   - tous les fichiers sources de votre projet, c'est-à-dire les fichiers d'extensions `.h` et `.c` ;
   - un fichier de description des dépendances de nom `Makefile` ;
   - un (ou trois) fichier(s) de texte (obtenu avec un simple éditeur de texte) de nom `niveau-n.txt` où vous remplacerez n par le numéro du niveau que vous avez réalisé (1, 2 ou 3) ; ce fichier pourra être vide ou contenir d'éventuelles informations destinées à expliquer les choix que vous aurez faits.

   Vos fichiers sources doivent être indentés et contenir des commentaires permettant de bien comprendre votre travail. Aucun autre fichier (objets, exécutable, données, résultats, etc.) ne doit être présent dans le répertoire.
3. Créez une archive au format ZIP de ce répertoire ; cette archive doit porter le même nom que le répertoire auquel est ajoutée l'extension `.zip` ; pour produire cette archive, dans le terminal, vous pouvez vous placer dans le répertoire parent du répertoire contenant vos fichiers sources et utiliser la commande (␣ représente un caractère espace) :
   `zip␣NOM-PRENOM-NUMETU.zip␣NOM-PRENOM-NUMETU/*`

Respectez bien toutes ces consignes car des tests et des vérifications automatiques seront effectués sur votre projet.

1.5 Oral

Au moment de l'oral, vous devrez être prêt à :
- faire fonctionner votre programme ;
- répondre aux questions de l'enseignant ;
- effectuer des tests ou des modifications demandées par l'enseignant.

1.6 Données de tests fournies

Pour pouvoir effectuer vos tests, nous vous fournissons des fichiers texte disponibles sur Moodle. Ces fichiers sont des œuvres littéraires françaises (*Le Tour du Monde en 80 Jours* de Jules Verne, *Madame Bovary* de Gustave Flaubert, *L'Avare* de Molière...) en version texte, avec encodage ISO 8859-1 [1]. Ces textes sont issus de l'Association des Bibliophiles Universels. [2]

Par ailleurs, pour ouvrir vos fichiers de résultats (produits à partir de vos programmes), nous vous fournissons le fichier `ouvrirLatin1.html` permettant d'ouvrir et lire correctement un fichier encodé en ISO 8859-1 (mais vous pouvez aussi utiliser un éditeur de texte classique en activant le bon encodage).

1. https://fr.wikipedia.org/wiki/ISO/CEI_8859-1
2. http://abu.cnam.fr/

2 Niveau 1 : statistiques générales sur un fichier texte

2.1 Module `listemots`

Le module `listemots` permet de gérer une liste de mots, chaque mot étant représenté par la structure de données privée suivante :

```
// Mot d'un texte
struct sMot {
    char *mot; // le mot (chaîne de caractères allouée dynamiquement)
    int taille; // nombre de caractères du mot
    int occurrences; // nombre d'occurrences du mot dans un texte
    struct sMot *suivant; // mot suivant dans le contexte d'une liste chaînée
};
```

Quant à la liste, elle sera implémentée par la structure de données privée suivante :

```
// Liste de mots
struct sListeMots {
    struct sMot *debut; // pointeur vers le premier mot de la liste
    struct sMot *fin; // pointeur vers le dernier mot de la liste
    int nbMots; // nombre de mots de la liste
};
```

Le type public `tListeMots` sera défini dans le fichier `listemots.h` de la façon suivante :

```
// Type permettant de manipuler une liste de mots
typedef struct sListeMots *tListeMots;
```

La figure 1 illuste un exemple de représentation mémoire d'une liste de deux mots.

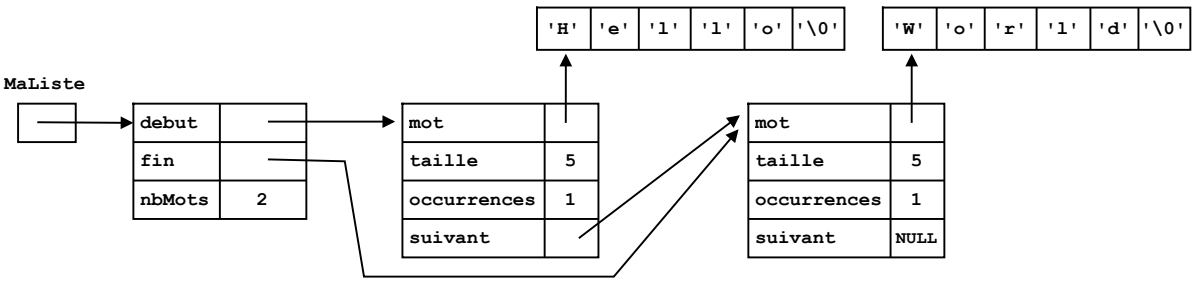

FIGURE 1 – *Représentation mémoire d'une liste de deux mots.*

Les fonctions privées suivantes devront être développées :

- `struct sMot *motCreer(char *mot, int occ)`
  crée un mot à partir d'une chaîne de caractères et de son nombre d'occurrences. La structure ainsi que son champ `mot` doivent être alloués dynamiquement (l'espace mémoire réservé pour le champ mot doit correspondre exactement à sa longueur). Tous les champs de la structure doivent être initialisés (champ `suivant` à `NULL`). Cette fonction doit renvoyer un pointeur vers le mot créé ou `NULL` en cas de problème d'allocation mémoire.
- `void motLiberer(struct sMot *mot)`
  libère tout l'espace mémoire occupé par la structure pointée par `mot`.

De plus, le module `listemots` devra comporter les fonctions publiques (déclarées dans le `.h`) suivantes :

- `tListeMots creerListeMots(void)`
  crée une liste vide de mot. La structure doit être allouée dynamiquement, les pointeurs `debut` et `fin` initialisés à `NULL`, et `nbMots` à 0. Cette fonction doit renvoyer la liste créée ou `NULL` en cas de problème d'allocation mémoire.
- `void detruireListeMots(tListeMots liste)`
  libère tout l'espace mémoire (y compris les mots) occupé par la liste.
- `int nombreDeMots(tListeMots liste)`
  renvoie le nombre de mots de la liste.
- `void ajouterMotAvecOcc(tListeMots liste, char *mot, int nbOcc)`
  ajoute un mot en fin de liste en initialisant le nombre d'occurrences du mot à `nbOcc`.
- `void ajouterMot(tListeMots liste, char *mot)`
  ajoute un mot en fin de liste en initialisant le nombre d'occurrences du mot à 1.
- `void ecrireListeMots(tListeMots liste, FILE *fich)`
  écrit tous les mots de la liste (en partant du début) dans le fichier désigné par `fich`. Seules les chaînes de caractères seront écrites (pas le nombre d'occurrences), chaque chaîne étant séparée par un caractère espace. L'écriture sera terminée par le caractère `\n` (nouvelle ligne). Sur l'exemple de la figure 1, le texte suivant sera écrit : `Hello World↩`

2.2 Test du module `listemots`

2.2.1 Compilation séparée

Créez le fichier source `testlistemots.c` contenant la fonction principale `main` afin de tester les différentes fonctions du module `listemots`. Dans cette fonction, vous créerez une liste, vous y ajouterez deux ou trois mots, vous afficherez son contenu (sur `stdout`) et enfin vous la détruirez.

Effectuez la compilation séparée grâce à l'utilitaire `make`. Pour cela, écrivez un fichier de description des dépendances `Makefile` et lancez les compilations en tapant : `make testlistemots`

2.2.2 Aide à la mise au point

Afin de vérifier que vos programmes n'ont pas de « fuites » mémoire, c'est-à-dire gèrent correctement l'allocation dynamique et la libération des zones en mémoire, un outil d'analyse dynamique peut vous aider. Par exemple, avec l'exécutable `testlistemots`, vous pouvez :

— sous Linux, utiliser Valgrind [3] avec la commande : `valgrind ./testlistemots`

— sous MacOS, utiliser Leaks avec la commande : `leaks --atExit --quiet -- ./testlistemots`

Dans les deux cas, vérifiez le nombre de « *leaks* » à la fin de l'affichage produit.

Il existe également des outils (*code sanitizers* [4]) qui détectent, au moment de l'exécution du programme, les erreurs d'accès à la mémoire comme le débordement d'un tableau. Sur les machines des salles de TP, quand vous invoquez `gcc`, vous pouvez ajouter l'option `-fsanitize=address` qui permettra l'affichage d'informations en cas d'erreur d'accès à la mémoire lors de l'exécution d'un programme.

2.3 Statistiques sur un fichier texte encodé en ISO Latin-1

L'objectif de ce programme est de calculer des statistiques basiques sur un fichier de texte, et de trouver la phrase la plus longue du texte. Le fichier en entrée sera encodé avec la norme ISO 8859-1 [5],

3. https://fr.wikipedia.org/wiki/Valgrind
4. https://en.wikipedia.org/wiki/Code_sanitizer
5. https://fr.wikipedia.org/wiki/ISO/CEI_8859-1

aussi appelée ISO Latin-1. Cet encodage reprend les caractères imprimables de l'US-ASCII, et ajoute de nombreux caractères de l'alphabet latin, dont la plupart des lettres accentuées. Chaque caractère est codé sur un octet.

2.3.1 Fonctions utilitaires

Afin de faciliter l'analyse des mots du texte, vous développerez les deux fonctions ci-dessous dans un module `utils` (qui sera également utilisé dans le niveau 2) constitué des fichiers `utils.h` et `utils.c`.

- `int caractereDunMot(unsigned char c)`
  renvoie 1 si le caractère `c` peut faire partie d'un mot, 0 sinon. Un caractère peut faire partie d'un mot si c'est une lettre minuscule ou majuscule, ou une lettre accentuée (valeur comprise entre 192 et 255 dans l'encodage ISO 8859-1), ou un chiffre, ou une apostrophe, ou un tiret.
- `int caractereFinDePhrase(unsigned char c)`
  renvoie 1 si le caractère `c` est un point, un point d'exclamation ou un point d'interrogation, 0 sinon.

2.3.2 Fonction principale

Vous écrirez la fonction principale du programme de statistiques dans le fichier `statstxt.c` qui utilisera les modules `listemots` et `utils`. Vous compléterez le `Makefile` précédent en ajoutant la cible `statstxt` pour produire l'exécutable. Ce dernier devra être appelé avec un argument, le nom du fichier à analyser (n'oubliez pas de vérifier le nombre d'arguments reçus). Exemple :

```
make statstxt
./statstxt le-tour-du-monde-en-80-jours-iso-8859-1.txt
```

Le fichier texte devra être lu entièrement, caractère par caractère. Le résultat de l'analyse doit permettre de calculer : le nombre de phrases contenues dans le texte, le nombre de mots, le nombre de caractères, la longueur moyenne d'un mot (en nombre de caractères) et la longueur moyenne d'une phrase (en nombre de mots). Il faudra également trouver la phrase la plus longue du texte (une phrase est considérée comme une liste de mots). Toutes ces statistiques seront écrites dans un fichier texte `statistiques.txt` selon le format détaillé ci-dessous.

Par exemple, voici le contenu du fichier `statistiques.txt` suite à l'exécution du programme sur le roman « Le Tour du Monde en 80 Jours » de Jules Verne (notez que la phrase la plus longue a été tronquée pour des raisons de place) :

```
Texte : le-tour-du-monde-en-80-jours-iso-8859-1.txt
Nombre de phrases : 5552
Nombre de mots : 67949
Nombre de caracteres : 426827
Longueur moyenne d'un mot : 4.94
Longueur moyenne d'une phrase : 12.24
Phrase la plus longue (123 mots) : En ce moment quelques auditeurs ...
```

Respectez scrupuleusement ce format (syntaxe des différentes lignes, précision de deux chiffres après la virgule pour les moyennes...) car des vérifications automatiques seront effectuées. Pour lire ce fichier `statistiques.txt`, vous pouvez soit l'ouvrir avec un éditeur de texte classique en activant l'encodage ISO 8859-1, soit passer par le fichier `ouvrirLatin1.html` que nous vous fournissons.

3 Niveau 2 : analyse de la fréquence des mots

3.1 Extension du module `listemots`

Dans ce niveau vous allez étendre le module précédent `listemots` avec les fonctions suivantes :

- `void ajouterOccurrenceMot(tListeMots liste, char *mot)`
  incrémente de 1 le compteur d'occurrences du mot concerné dans la liste. Si le mot n'est pas encore présent dans la liste, alors il est ajouté.
- `void ecrireCsvListeMots(tListeMots liste, char *nomFichier)`
  écrit dans un fichier texte, de nom `nomFichier`, tous les mots de la liste. Chaque mot doit être écrit sur une ligne selon la syntaxe suivante : *mot*, *taille*, *nombre-occurrences*
  Exemple :

  ```
  chanteur, 8, 2
  cuisinier, 9, 6
  ```

- `void supprimerPetitsMots(tListeMots liste, int tailleMin)`
  supprime tous les mots de la liste dont la taille est strictement inférieure à `tailleMin`. Les mots concernés doivent être retirés de la liste et la mémoire associée libérée.

3.2 Analyse de la fréquence des mots

Créez la fonction principale pour l'analyse de la fréquence des mots d'un texte dans le fichier `analysermots.c` qui utilisera les modules `listemots` et `utils`. Vous compléterez le `Makefile` précédent en ajoutant la cible `analysermots` pour produire l'exécutable. Ce dernier devra être appelé avec deux arguments (faites les vérifications usuelles sur le nombre d'arguments reçus) : le nom du fichier à analyser et la taille minimale des mots que nous souhaitons considérer. Exemple :

```
make analysermots
./analysermots le-tour-du-monde-en-80-jours-iso-8859-1.txt 4
```

De manière similaire au niveau 1, le fichier texte devra être lu caractère par caractère. Dès qu'un nouveau mot est détecté, ce dernier doit être ajouté à la liste s'il n'est pas déjà présent, sinon son nombre d'occurrences doit être mis à jour (incrémenté de 1).

Une fois tout le texte analysé, deux fichiers seront produits en sortie (selon le format défini dans la fonction `ecrireCsvListeMots`) :

— `liste_mots_01.txt` qui contiendra la liste complète des mots du texte avec leur nombre d'occurrences ;
— `liste_mots_02.txt` qui contiendra uniquement les mots dont la taille est supérieure ou égale au deuxième argument passé au programme.

N'oubliez pas de libérer la mémoire à la fin de votre programme.

Comme précédemment, vous pouvez lire les fichiers produits avec un éditeur de texte en activant l'encodage ISO 8859-1, ou en utilisant le fichier `ouvrirLatin1.html` que nous vous fournissons.

4 Niveau 3 : tri liste de mots

4.1 Extension du module `listemots`

Dans ce niveau vous allez étendre le module précédent `listemots` avec les fonctions suivantes :

- `void lireCsvListeMots(tListeMots liste, char *nomFichier)`
  lit un fichier texte, de nom `nomFichier`, qui contient un mot par ligne, selon le format produit par la fonction `ecrireCsvListeMots` (c'est-à-dire *mot*, *taille*, *nombre-occurrences*). Les mots lus seront ajoutés à la liste déjà créée et passée en paramètre.
- `void trierListeNombreOccurrences(tListeMots liste)`
  trie la liste chaînée des mots par ordre décroissant du nombre d'occurrences. L'algorithme de tri n'est pas imposé. La performance n'étant pas un objectif, vous pouvez opter pour un tri simple du type tri par sélection [6], tri à bulles [7], ou tri par insertion [8].

4.2 Tri de la liste de mots

Créez le fichier `triermots.c` contenant la fonction `main`. Vous compléterez le `Makefile` précédent en ajoutant la cible `triermots` pour produire l'exécutable. Ce dernier devra être appelé avec un argument : le nom du fichier contenant la liste à trier (fichier issu de la section 3.2). Exemple :

```
make triermots
./triermots liste_mots_02.txt
```

Une fois la liste de mots triée par ordre décroissant du nombre d'occurrences, elle devra être écrite dans un fichier `liste_mots_03.txt` (toujours selon le même format défini dans la fonction `ecrireCsvListeMots`).

6. https://fr.wikipedia.org/wiki/Tri_par_selection
7. https://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles
8. https://fr.wikipedia.org/wiki/Tri_par_insertion