Examen - ING1 GI - 2019/2020
Programmation C

Modalités

- Durée : 3 heures
- Toutes vos affaires (sacs, vestes, trousse, etc.) doivent être placées à l'avant de la salle.
- Aucun document papier n'est autorisé.
- Aucune sortie n'est autorisée avant une durée incompressible de 1 heure.
- Aucun déplacement n'est autorisé.
- **Aucune question au professeur n'est autorisée**. Si le sujet ne vous semble pas clair, à vous d'expliquer dans des commentaires pourquoi est ce que ça ne vous semble pas clair, et quelles modifications vous effectuez pour réaliser l'exercice.
- Aucun échange, de quelque nature que ce soit, n'est possible.
- Les différentes fonctions/procédures demandées seront toutes à tester dans un **main**.
- Vous traiterez chaque exercice dans un dossier différent.
- Le barème est **indicatif**, il intègre pour chaque question un test dans un main.
- La présence d'un Makefile et des commentaires doxygens sont un plus, et seront appréciés.
- La lisibilité du code et les commentaires seront pris en compte dans la notation, de même que la séparation des différentes procédures/fonctions dans des fichiers d'en-tête (.c .h).
- **Tout code qui ne compile pas ne sera pas corrigé et entraînera une pénalité de 50%. Si une partie de votre code ne compile pas, commentez-le. Ceci implique évidemment que vous compiliez au fur et à mesure!**
- Chaque **warning** de compilation entraînera une **pénalité de 15%**.
- Tout ce qui sera dans le dossier **rendu** correspondra à votre rendu, et sera récupéré à la fin de l'examen. Je vous conseille vivement de travailler directement dans ce dossier.

Exercice 1 : Déchiffrage par analyse fréquentielle (1,5 + 1 + 2,5 + 1 = 6 pts)

L'analyse fréquentielle est l'étude de la répartition des lettre dans un texte. Elle facilite le déchiffrement de messages chiffrés par substitution (ex : code de César) en se basant sur le fait que certaines lettres ou combinaisons de lettres n'apparaissent pas aussi souvent que d'autres dans les langages : en anglais, E est la lettre la plus utilisée, alors que Z l'est beaucoup moins.

Vous avez à disposition deux textes anglais qui ont été cryptés par la méthode de César, avec un décalage différent pour chaque texte : `texte-crypte.txt` et `texte2-crypte.txt`.
L'objectif de l'exercice est de pouvoir créer un nouveau fichier contenant le texte en clair.

Les étapes pour décrypter sont les suivantes :

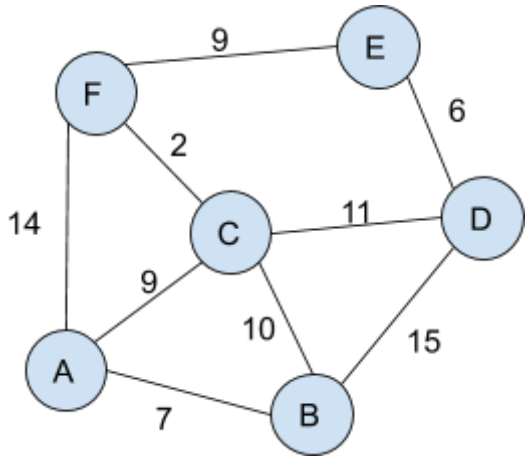
1. Créer un histogramme de fréquence d'apparition des lettres de l'alphabet (sans différencier en fonction de la casse).
2. Calculer le décalage du cryptage *d*. On sait que la fréquence la plus élevée en anglais correspond à la lettre 'e'.
3. Créer un fichier contenant le texte en clair. Pour faire cela, il suffit d'annuler le décalage du cryptage de César en décalant de *-d*.

Question 1 : Écrire la procédure `histogramme` qui permet de compter les fréquences d'apparition de chacune des lettres d'un fichier donné.
Question 2 : Écrire la fonction `calculerDecalage` qui permet de calculer le décalage *d* qu'il y a eu lors du cryptage de César.
Question 3 : Écrire la procédure `decrypterTexte` qui permet de recréer le fichier texte en clair.
Question 4 : Écrire une fonction `main` qui prendra en premier argument le nom du fichier à décrypter et en deuxième argument le nom du fichier résultat.
Exemple d'appel du programme :

```
./programme  texte2-crypte.txt  texte2-origine.txt
```

Exercice 2 : Chemin le plus court (1 + 1,5 + 1,5 + 2,5 + 3 + 1,5 + 1,5 + 1,5 = 14 pts)

On souhaite pouvoir déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Un réseau routier est défini par un ensemble de villes et un ensemble de routes possibles entre ces villes. On connaît la distance pour chacune de ces routes.
Voici un exemple de réseau routier à 6 villes {A,B,C,D,E,F} :



Afin de modéliser informatiquement ce réseau routier, nous allons utiliser un tableau à deux dimensions qui répertorie l'ensemble des routes directes possibles entre les villes. La ligne correspond à la ville de départ et la colonne correspond à la ville d'arrivée. La valeur dans la case correspond à la distance entre ces deux villes. S'il n'y a pas de chemin, la valeur 0 est définie.
Par rapport à l'exemple donné, voici le tableau que nous obtenons :

	A	B	C	D	E	F
A	0	7	9	0	0	14
B	7	0	10	15	0	0
C	9	10	0	11	0	2
D	0	15	11	0	6	0
E	0	0	0	6	0	9
F	14	0	2	0	9	0

En théorie des graphes, l'algorithme de *Dijkstra* sert à résoudre le problème du plus court chemin. C'est un algorithme itératif qui construit progressivement un chemin optimal. Tout au long de l'algorithme, nous avons besoin de connaître pour chaque ville :

- la **distance cumulée** qui est la somme des routes pour aller de la ville courante à la ville de départ ;
- la **ville précédente** qui est la dernière ville par laquelle on est passé pour obtenir la distance cumulée actuelle ;
- si la ville a déjà été **traitée** ou non.

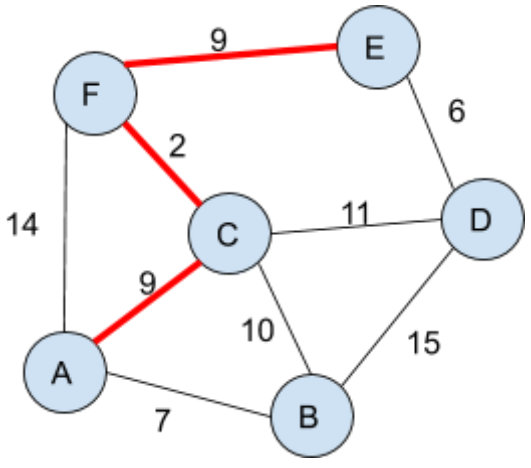
Au départ, on considère que les distances de chaque ville à la ville de départ (distance cumulée) sont infinies, sauf pour la ville de départ pour laquelle la distance est nulle.
Au cours de chaque itération :

1. Parmi les villes qui n'ont pas encore été traitées, on cherche la ville *vMin* dont la distance cumulée est la plus petite ;
2. Pour chaque ville *vVoisine* qui lui est directement reliée (valeur non nulle dans la ligne de *vMin* du tableau à 2 dimensions)
    a. Si la distance cumulée de *vVoisine* est plus grande que la distance cumulée de *vMin* + la longueur de la route entre *vMin* et *vVoisine*, alors on garde ce chemin là puisqu'il est plus intéressant. On met à jour la distance cumulée et la ville précédente.
3. Après avoir traité toutes les villes voisines de *vMin*, on marque *vMin* comme ayant déjà été traitée.

Afin d'être sûr d'avoir tout traité, le nombre d'itération correspond au nombre de villes.
Par rapport à l'exemple donné précédemment, en prenant A comme la ville de départ et E la ville d'arrivée, nous obtenons (après les *n* itérations) :

Ville	Distance cumulée	Ville précédente
A	0	
B	7	A
C	9	A
D	20	C
E	20	F
F	11	C

Afin de reconstituer le chemin le plus court entre le départ et l'arrivée, il suffit de partir de la d'arrivée et de prendre la ville précédente, ainsi de suite jusqu'à arriver à la ville de départ.
Par rapport à l'exemple précédent, on est arrivée à E en passant par F, et avant par C, et avant par A. Sur le schéma, nous avons donc ce chemin :



Nous allons donc implémenter cette méthode. Les questions sont là pour vous aider à découper correctement le code et vous rappeler ce qui doit être fait à chaque étape.

Question 1 : Définir une structure `sVille`, définie comme suit :

Enregistrement `sVille`

indice : entier *// Correspond à l'indice dans le tableau à 2 dimensions*
nom : chaîne de caractères *// Nom de la ville*
distanceCumulee : entier
iVillePrecedente : entier *// Correspond à l'indice dans le tableau de la ville précédente*
estTraitee : booléen

Fin Enregistrement

Question 2 : Écrire la fonction `sVille* initVilles(int n)` qui crée un tableau de *n* villes et initialise chaque champ de la structure. Le *nom* de la ville sera saisie par l'utilisateur. L'*indice* de la ville correspondra à l'indice i de la ville dans le tableau de ville créé. A l'initialisation, les champs *distanceCumulee* et *iVillePrecedente* sont initialisées à -1, et *estTraitee* est à FAUX.

Question 3 : Écrire la fonction `int** reseauRoutier(int n)` qui crée et initialise le tableau à deux dimensions. Les valeurs du tableau (distance entre deux villes) seront saisies par l'utilisateur.

Question 4 : Écrire la fonction `int villeDistMinNonTraitee(sVille* tabVilles, int n)` qui renvoie l'indice de la ville dont la distance cumulée est la plus petite. On ne considère que les villes qui n'ont pas encore été traitées.

Question 5 : Écrire la procédure `void majDistCumule (sVille* tabVilles, int** reseauRoutier, int n, int idMin)`. Elle prend en paramètre le tableau de villes *tabVilles*, le réseau routier *reseauRoutier*, le nombre de villes *n* et l'indice de la ville *idMin* dont la distance cumulée est la plus petite.
Pour chaque ville *vVoisine* qui est directement reliée à la ville *idMin* (valeur non nulle dans la ligne de *idMin* du *reseauRoutier*) :

- si la distance cumulée de *vVoisine* est plus grande que la distance cumulée de *idMin* + la longueur de la route entre *idMin* et *vVoisine*, alors on garde ce chemin. On met à jour :
    - la distance cumulée de *vVoisine,* qui devient la distance cumulée de *idMin* + la longueur de la route entre *idMin* et *vVoisine*
    - la ville précédente de *vVoisine*, qui devient *idMin*.

Question 6 : Écrire la procédure `void dijkstra (sVille* tabVilles, int** reseauRoutier, int n)`. Elle prend en paramètre le tableau de villes tabVilles, le réseau routier reseauRoutier et le nombre de villes n. Elle fait *n* itération. A chaque itération, il faut :

1. chercher la ville *vMin* à l'aide de la fonction *`villeDistMinNonTraitee`*
2. mettre à jour les distances cumulées à l'aide de la procédure *`majDistCumule`*
3. changer la valeur du champ *estTraitee* de la ville *vMin* à VRAI.

Question 7 : Écrire la méthode `afficheCheminPlusCourt` qui permet d'afficher le chemin le plus court, c'est à dire la ville de départ, les villes intermédiaires (dans le bon ordre) et la ville d'arrivée.

Question 8 : Écrire le programme principale (main) qui :

- demande le nombre de villes du réseau routier
- initialise chaque ville
- initialise le réseau routier
- demande la ville de départ et la ville d'arrivée
- applique la méthode de dijkstra
- affiche le chemin le plus court entre la ville de départ et d'arrivée.

Vous avez à disposition deux fichiers `dijkstra-1.txt` et `dijkstra-2.txt` qui contiennent ce qui est à saisir par l'utilisateur. Vous pouvez les utiliser (avec une redirection de l'entrée standard) pour vous faciliter la saisie. Exemple d'appel du programme :

```
cat dijkstra-1.txt | ./programme
```