Programmation C
TP n° 8 : Chaînes de caractères et paramètres de *main*

Instructions :

1. Vous pouvez utiliser les fonctions de la bibliothèque standard vues en cours, notamment celles définies dans `ctype.h` et `string.h`.
2. À chaque fois que vous faites un malloc, assurez-vous que la mémoire ait été correctement allouée avec `assert`.
3. Veillez à bien tester chacune des fonctions.
4. Le TP est à rendre le 2 avril sur Moodle.

Exercice 1 : Mot le plus long

Écrivez un programme `longest.c` qui affiche l'argument le plus long en nombre de caractères parmi ceux passés à `main`. Par exemple, `./longest␣a␣++␣Z_zz␣a08o␣pP` affiche `Z_zz`.

Exercice 2 : Ordre lexicographique

On souhaite écrire un programme qui affiche le plus grand des arguments passés à `main` selon l'ordre lexicographique (l'ordre du dictionnaire : a est plus petit que ab, ou encore a est plus petit que bcd, etc).

1. Écrivez une fonction `char *normalise(const char *s)`. Cette fonction doit allouer, construire et renvoyer l'adresse d'une nouvelle chaîne formée de la suite de toutes les lettres de la chaîne d'adresse `s`, en excluant tous les caractères qui ne sont pas des lettres et où toutes les majuscules sont passées en minuscules.
2. Complétez le `main` pour afficher la valeur voulue.

Exercice 3 : Concaténation

Dans cet exercice, par *mot* d'une chaîne, on entend toute suite maximale de caractères adjacents dans la chaîne et tous différents de l'espace (`' '`). Par exemple, la suite des mots de `" a aa ba a bbbb "` est `"a"`, `"aa"`, `"ba"` `"a"` et `"bbbb"` Notez qu'il peut y avoir plusieurs espaces entre deux mots, et des espaces avant le premier ou après le dernier. Même non vide, une chaîne peut ne contenir aucun mot – si elle ne contient que des espaces.

1. Écrivez une fonction `char *concat_words(size_t n, char *tab[])` allouant, construisant et renvoyant l'adresse d'une nouvelle chaîne formée de la concaténation de toutes celles dont les adresses sont les éléments de `tab`, en séparant chaque couple de chaînes successives par une unique espace. Testez votre fonction en concaténant les paramètres de `main`.
2. Écrivez une fonction `int nb_words(const char *s)` renvoyant le nombre de mots de la chaîne d'adresse `s`. Par exemple, si cette chaîne est `" a aa ba a bbbb"`, la fonction doit renvoyer `5`.
3. Écrivez une fonction `char **split(const char *s, int *n)`. Cette fonction doit allouer un vecteur d'adresses de chaînes. La taille du vecteur sera `nb_words(s)`. Autrement dit, le vecteur doit être composé de `nb_words(s)` élément de type `char *`. Dans le i-ème élément du vecteur vous devez mettre l'adresse d'une copie du i-ème mot de `s`. Vous devez mettre la valeur `nb_words(s)` à l'adresse `n`. La fonction retourne l'adresse du vecteur. Par exemple si l'on a :

```c
int n;
char **v = split("ala  ma  kota ", &n);
```

alors dans `v` nous récupérons l'adresse d'un vecteur de trois éléments, où l'on a
- à `v[0]` l'adresse d'une chaîne `"ala"`,
- à `v[1]` l'adresse d'une chaîne `"ma"`,
- à `v[2]` l'adresse d'une chaîne `"kota"`.

Dans `n` on récupère la valeur 3. N'oubliez pas d'allouer la mémoire pour chaque mot dont l'adresse est stockée dans le vecteur.

4. Testez votre fonction en vérifiant qu'en appliquant `split` à une chaîne, puis en appliquant `concat_words` au tableau résultant, on obtient bien une chaîne contenant la même suite de mots que la chaîne initiale.

Exercice 4 : Algorithmique du texte

Dans cet exercice, on travaille sur une représentation de l'ADN sous forme de chaînes de caractères composées des caractères a, c, g, t. On nomme *mutation* une différence d'une chaîne par rapport à une autre de même taille. Par exemple, dans `"acca"`, `"cc"` à l'indice 1 est une mutation de longueur 2 par rapport à la chaîne `"aaaa"`. Une mutation est représentée par la structure suivante :

```c
typedef struct {
  size_t indice;
  size_t len;
} mutation;
```

1. Écrivez une fonction `int occurences(const char *s, const char *sub)` qui renvoie le nombre d'occurences de la chaîne d'adresse `sub` dans celle d'adresse `s`. Par exemple `"aa"` a trois occurrences dans `"aaacaa"` (aux positions `0`, `1` et `4`).
2. Écrivez une fonction `mutation diff(const char *s, const char *t)` qui renvoie la première mutation de `t` par rapport à `s`. Par exemple, si `m = diff("acca", "aaaa")`, alors `m.indice = 1` et `m.len = 2`. Si les chaînes sont identiques, la mutation renvoyée est de longueur nulle et d'indice quelconque.
3. En se servant de cette fonction, écrivez `mutation longest(const char *s, const char *t)` qui renvoie la première plus longue mutation de `t` par rapport à `s`. Par exemple, si `m = longest("atcgatatt", "aaagccata")`, alors `m.indice = 1` et `m.len = 2`. Pour cela, utilisez `diff` sur `s` et `t` puis à nouveau sur les suffixes de ces chaînes commençant après la première mutation pour trouver la deuxième, et ainsi de suite.
4. Écrivez une fonction `char *longest_string(const char *s, const char *t)` qui renvoie l'adresse d'une chaîne qui est une copie de la plus longue mutation de `t` par rapport à `s`. Utilisez la fonction `strncpy`.
5. Écrivez des fonction `diff2` et `longest2` qui acceptent deux chaînes de tailles diférentes. On considère le fait qu'une chaîne s'arrête avant l'autre comme une mutation. Par exemple, si `m = diff2("acattta", "acg") = 2`, alors `m.indice = 2` et `m.len = 5`. La mutation commence au dernier caractère, `'g'`.
6. Écrivez une fonction `int prefix(const char *s, const char *t)` qui renvoie 1 si `s` est un préfixe de `t`, -1 si c'est l'inverse, et 0 si aucune chaîne n'est préfixe de l'autre. Faites-le sans utiliser de boucle en vous servant de la fonction `diff`.