Programmation en C

Examen du 9/01/2014 - 3h

Bien qu'inutiles, les notes de cours sont autorisées, pas les livres. Dans une certaine mesure, la correction tiendra compte de la lisibilité du code produit et de son efficacité en termes de temps d'exécution et de quantité de mémoire utilisée.

Le sujet est sur 4 pages. Chaque question peut être traitée indépendamment. Prenez le temps de bien comprendre chaque définition, et cherchez avant tout la méthode correcte la plus simple – il est possible d'écrire chaque fonction de manière naturelle avec très peu de variables.

I - Chaînes de caractères

Contraintes. Deux contraintes sont à respecter dans toute cette partie :

1. Les fonctions ci-dessous ne doivent ni déclarer, ni allouer de nouveaux tableaux, et ne peuvent appeler aucune autre fonction.
2. Chaque chaîne ne pourra être lu qu'une fois au plus – pas de retours en arrière, pas de `strlen`.

Notations. Pour toute chaîne de caractère w et pour toute position i dans w, on note w_i le caractère de w de position i – comme dans un tableau en C, ces positions sont numérotées à partir de 0.

Exercice 1

Etant données deux chaînes s, t, appelons *occurrence commune* dans s et t chaque position i vérifiant les deux conditions suivantes : i est à la fois une position dans s et dans t ; et, s_i et t_i sont égaux. Par exemple, les deux chaînes suivantes ont trois occurrences communes (1, 3, 4):

```
"y a z b c x"
"x a y b c z d e f"
```

Ecrire `int communes (char *s, char *t)` renvoyant le nombre d'occurrences communes dans les chaînes d'adresses `s`, `t`.

Exercice 2

Etant données deux chaînes s, t, appelons *occurrence disjointe* dans s ou t chaque position qui est une position dans s ou une position dans t, mais qui n'est pas une occurrence commune (au sens de l'exercice 1). Par exemple, les deux chaînes suivantes ont six occurrences disjointes (0, 2, 5, ainsi que 6, 7, 8 qui ne sont des positions que dans la seconde chaîne) :

```
"y a z b c x "_ _
"x a y b c z d e f"
```

Ecrire `int disjointes (char *s, char *t)` renvoyant le nombre d'occurrences disjointes dans les chaînes d'adresses `s`, `t`. N'oubliez pas de respecter les contraintes ci-dessus – en particulier, cette fonction ne doit pas appeler la précédente.

Exercice 3

Etant données trois chaînes s, t, u, on dira que s est un *mélange de t et u* si la longueur de s est celle de la plus grande des deux chaînes, et si pour chaque position i dans s, l'une au moins des conditions suivantes est vérifiée :

> i est une position dans t, et s_i est égal à t_i, ou,
>
> i est une position dans u, et s_i est égal à u_i.

Voici par exemple tous les mélanges de `"xy"` et `"abcd"` – à la position 0, peut avoir `'x'` ou `'a'` ; à la position 1, on peut avoir `'y'` ou `'b'` ; enfin, 2, 3 ne sont des positions que dans `"abcd"`, et ne peuvent donc contenir que `'c'`, `'d'` :

`"abcd" "xbcd" "aycd" "xycd"`

Ecrire `int est_melange(char *s, char *t, char *u)` renvoyant `1` si la chaîne d'adresse `s` est un mélange des chaînes d'adresses `t`, `u`, et renvoyant `0` sinon.

Exercice 4

On dira que s est un *croisement* de t et u si s est de même longueur que u, et si pour un certain entier k inférieur où égal aux longueurs de t et de u, on a

$$s = t_0 \dots t_{k-1}\, u_k \dots u_{n-1}$$

Voici par exemple tous les croisements de `"xyz"` et `"abcde"` – la plus petite chaîne est de longueur 3, et chaque croisement est bien, pour un certain k entre 0 et 3 inclus, formé des k premiers caractères de `"xyz"` suivi de `"abcde"` privé de ses k premiers caractères :

`"abcde" "xbcde" "xycde" "xyzde"`

Autre exemple, tous les croisements de `"abcde"` et `"xyz"` – là encore, chaque croisement est, pour un certain k entre 0 et 3 inclus, formé des k premiers caractères de `"abcde"` suivi de `"xyz"` privé de ses k premiers caractères :

`"xyz" "ayz" "abz" "abc"`

Ecrire `int est_croisement(char *s, char *t, char *u)` renvoyant `1` si la chaîne d'adresse `s` est un croisement des chaînes d'adresses `t`, `u`, et renvoyant `0` sinon. Attention au cas où la seconde chaîne est plus courte que la première.

Exercice 5

On dira que s est une *démultiplication* de $t = t_0 \dots t_{n-1}$ si s est de la forme

$$s = t_0\, t_0 \dots t_0\ t_1\, t_1 \dots t_1 \dots\ t_{n-1}\, t_{n-1} \dots t_{n-1}$$

autrement dit, s peut être obtenu à partir de t en remplaçant chaque caractère par un ou plusieurs exemplaires de ce caractère. Par exemple, `"aaabbccc"` est une démultiplication de `"aabcc"`. En revanche, `"aabb"` *n'est pas* une démultiplication de `"aaab"`.

Ecrire `int est_demultiplication(char *s, char *t)` renvoyant `1` si la chaîne d'adresse `s` est une démultiplication de celle d'adresse `t`, et renvoyant `0` sinon. Vérifiez bien l'algorithmique. Notez qu'il faut nécessairement deux compteurs, un pour chaque chaîne.

Exercice 6 (Problème)

On considère à présent une chaîne t formant un texte découpé en lignes. Chaque ligne, y compris la dernière, se termine par un unique `'\n'`. Il peut y avoir des lignes vides - qui, mêmes vides, comptent comme des lignes.

Chaque ligne de t est elle-même découpée en mots : l'espace est considéré comme l'unique séparateur de mots. Il peut y avoir zéro ou plusieurs espaces avant le premier mot d'une ligne ou après le dernier mot d'une ligne, ainsi que un ou plusieurs espaces entre deux mots.

On dira que t forme un *abécédaire boustrophédonique* si la suite de lettres ci-dessous forme exactement la suite des lettres de l'alphabet `'a'` ... `'z'` :

- la suite des premières lettres des mots de la $1^{\text{ère}}$ ligne, suivi de,
- la suite inversée des premières lettres des mots de la 2^{nde} ligne, suivi de,
- la suite des premières lettres des mots de la $3^{\text{ème}}$ ligne, suivi de,
- la suite inversée des premières lettres des mots de la $4^{\text{ème}}$ ligne, suivi de,
- etc.

Autrement dit, la texte a, par exemple, la structure suivante (en remplaçant les points par des caractères arbitraires) :

```
a... b.... c....d...
i.... h.... g... f... e...
j... k...
n... m... l...
```

etc. Ecrire `int boustr_abc(char *s)` renvoyant `1` la si le texte d'adresse `t` est un abécédaire boustrophédonique, et renvoyant `0` sinon.

Cette fonction nécessite de mémoriser un peu plus d'information que les précédentes. Elle doit encore être écrite en respectant les contraintes, dont celle de la lecture unique ; à défaut, vous pouvez proposer une solution partielle dans laquelle le texte sera lu, au plus, deux fois en tout.

II - Listes chaînées et récurrence

On considère le type usuel permettant de représenter en C des listes chaînées d'entiers, avec les conventions habituelles :

```
struct cell {
  int clef;
  struct cell *suiv;
};
```

Les deux fonctions ci-dessous doivent être écrites par récurrence pure : pas de boucles, pas de variables statiques. Chaque fonction ne peut appeler qu'elle-même, et `free` Aucune ne nécessite d'allocation mémoire. Attention aux cas limites, à l'ordre des opérations et à la syntaxe des appels récursifs.

Exercice 7

Ecrire `struct cell *supprimer_debut(int k, struct cell *pc)`. Cette fonction doit renvoyer la liste obtenue en supprimant les `k` premières cellules de `pc` – ou *toutes* les cellules de `pc` si cette liste contient moins de `k` cellules. Tout l'espace mémoire alloué pour les cellules supprimées devra être libéré.

Exercice 8

Un peu plus difficile, écrire

```
int supprimer_fin(int k, struct cell *pc)
```

Cette fonction doit supprimer les `k` dernières cellules de `pc` – ou *toutes* les cellules de `pc` si cette liste contient moins de `k` cellules. Là encore, l'espace mémoire alloué pour les cellules supprimées devra être libéré. En retour, cette fonction doit renvoyer le nombre de cellules effectivement supprimées.

Noter que la valeur de retour d'un appel récursif est précisément ce qui permet à un appel courant de savoir s'il doit supprimer sa première cellule ou non.