Programmation en C

Examen du 17/06/2010 - 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.

Chaque question peut être traitée indépendamment. Ne perdez pas de temps à écrire des `main` pour les fonctions demandées, ils ne seront pas pris en compte.

I - Matrices

Appelons *densité* d'une matrice d'entiers le nombre maximal de valeurs successives non nulles sur une même ligne ou sur une même colonne. Par exemple, la matrice suivante est de densité 3 - une colonne contient exactement 3 valeurs successives non nulles (3, 6, 5), et chaque ligne et chaque colonne contient au plus 3 valeurs successives non nulles :

$$\begin{pmatrix} 2 & 0 & 3 & 2 & 0 \\ 0 & 3 & 6 & 0 & 2 \\ 5 & 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 \end{pmatrix}$$

Ecrire `int densite(int m[H][L])` où `H`, `L` sont deux constantes entières quelconques définies en début de programme, renvoyant la densité de `m`. Vous pouvez déclarer dans cette fonction des tableaux auxiliaires (en fait, un seul suffit), mais le contenu de la matrice ne devra être lu qu'une seule fois - vraiment une seule, à vous de trouver comment.

II - Nouvel abrégé d'abrégé de littérature potentielle

Dans chacun des exercices suivants, on appelle par convention *lettre* tout caractère minuscule non accentué (compris entre `'a'` et `'z'` inclus). Tous les autres caractères seront considérés comme des séparateurs de mots, et appelés *séparateurs*. Pour éviter de trop alourdir le code, il est conseillé de se servir de la fonction auxiliaire suivante :

```
int lettre(char c) {
   return 'a' <= c && c <= 'z';
}
```

Il peut y avoir plusieurs séparateurs entre deux mots. Un texte peut être vide, ou ne contenir que des séparateurs, ou que des lettres. Il peut commencer ou finir par n'importe quel caractère.

Exercice 1

Une *boule de neige* est un texte dans lequel le premier mot est composé d'une seule lettre, le second de deux lettres, le troisième de trois lettres, etc. Par exemple le texte `"l'un des cinq vieux chiens aboyait gaiement."` est une boule de neige.

Ecrire `int boule_de_neige(char *s)` renvoyant `1` si le texte stocké en `s` est une boule de neige, et `0` sinon. Le texte ne devra être lu qu'une fois au plus.

Exercice 2

Un texte est *localement palyndromique* si chacun de ses mots forme un palyndrome (se lit de la même manière dans un sens et dans l'autre). Par exemple, `"non, elle alla ressasser...  ici."` est localement palyndromique.

Ecrire `int loc_pal(char *s)` renvoyant `1` si le texte stocké en `s` est localement palyndromique, et `0` sinon.

Exercice 3

Ecrire une fonction `char *mots_longs(char *s, int n)` effectuant le traitement suivant. Cette fonction doit construire un nouveau texte consistant en tous les mots du texte stocké en `s` de longueur au moins `n`. Ces mots seront séparés par des espaces simples dans le texte resultat, sans espace avant le premier mot, ni après le dernier. Si par exemple le texte initial est `"...  hum !... non, un texte vraiment pas trop court."` et si `n` vaut `4`, le texte résultat sera `"texte vraiment trop court"`.

Le texte initial ne devra être lu que deux fois en tout. La texte résultat devra être stocké dans une zone mémoire allouée (exactement de la taille suffisante pour son stockage) dont la fonction renverra l'adresse.

Exercice 4

Appelons *sous-texte* d'un texte tout autre texte dont la suite de mots est une sous-suite de la suite de mots du texte initial. Par exemple, `"ceci est...  un sous-texte !"` est sous-texte de `"ceci est un texte contenant bien sur un sous-texte..."`, car la suite de mots :

> `"ceci", "est", "un", "sous", "texte"`

est effectivement une sous-suite de la suite de mots :

> `"ceci", "est", "un", "texte", "contenant", "bien","sur", "un", "sous", "texte"`

Ecrire une fonction `int sous_suite(char *s, char *t)` renvoyant `1` si le texte stocké en `t` est sous-suite de celui stocké en `s`, et `0` sinon. Les relectures de fragments de `s`, `t` devront être réduites au minimum.

III - Listes chaînées

Exercice

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 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, si nécessaire, `free`. Attention aux cas limites.

Question 1

Ecrire par récurrence une fonction

```
struct cell *filtrer(struct cell *pc, int n)
```

Cette fonction doit extraire de `pc` la sous-liste formée de toutes les clefs de valeur supérieure ou égale à `n`. Toutes les autres cellules seront libérées par `free`. Si par exemple `pc` contient la suite de clefs $\langle 3, 0, 2, 1, 3, 0, 4, 6 \rangle$ et `n` vaut 2, la liste résultante aura pour suite de clefs $\langle 3, 2, 3, 4, 6 \rangle$. Cette fonction ne doit pas allouer de nouvelles cellules, ni modifier de clefs. Elle devra renvoyer l'adresse de la première cellule de la liste résultante.

Question 2

Par convention, la *position* d'une cellule dans une liste est le nombre de cellules qui la précèdent, e.g. la première cellule d'une liste est à la position zéro. Ecrire par récurrence une fonction

```
struct cell *echanger_paires(struct cell *pc)
```

En modifiant seulement le chainage de `pc`, cette fonction doit échanger avec son successeur chaque cellule de `pc` de position paire ayant un successeur. Si par exemple `pc` contient la suite de clefs $\langle n_0, n_1, n_2, n_3, n_4, n_5, n_6 \rangle$, la liste résultante aura pour suite de clefs $\langle n_1, n_0, n_3, n_2, n_5, n_4, n_6 \rangle$.

Cette fonction ne devra allouer ou libérer aucune cellule, ni modifier de clefs. Elle devra renvoyer l'adresse de la première cellule de la liste résultante.