Programmation en C

Examen du 6/01/2016 - 3h

Bien qu'inutiles, les notes de cours sont autorisées, pas les livres. Chaque exercice peut être traité indépendamment. 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.

Ne perdez pas de temps à écrire des `main` pour les fonctions demandées : le code des fonctions suffit.

I - Boucles et chaînes

Par *chaîne*, on entend une chaîne de caractères usuelle, de contenu quelconque – lettres, symboles, chiffres, espaces ...

Appelons *tranche* d'une chaîne s toute sous-chaîne non vide s formée de caractères identiques, et qui n'est extensible ni vers la gauche, ni vers la droite. Par exemple, la chaîne `"aaabbaaaacdd"` contient successivement les tranches `"aaa"`, `"bb"`, `"aaaa"`, `"c"` et `"dd"`.

Exercice 1

Ecrire `int nbr_tranches(char *s)` renvoyant le nombre de tranches de la chaîne d'adresse `s`. Par exemple, si la chaîne est `"aaabbaaaacdd"`, la valeur renvoyée sera `5`. La chaîne ne devra être parcourue qu'une et une seule fois.

Exercice 2

Ecrire `int tranche_max(char *s)` renvoyant `0` si la chaîne d'adresse `s` est vide, et sinon, renvoyant la longueur de sa plus grande tranche. Par exemple, si la chaîne est `"aaabbaaaacdd"`, la valeur renvoyée sera `4` (soit la longueur de la tranche `"aaaa"`). La chaîne ne devra être parcourue qu'une et une seule fois.

Exercice 3

On dira qu'une chaîne est *à tranches incrémentales* si chaque tranche qui n'est pas la première contient exactement un caractère de plus que la précédente. Par exemple, toute chaîne contenant au plus une tranche est à tranches incrémentales. La chaîne `"aabbbaaaaccccc"` est à tranches incrémentales.

Ecrire `int tranches_incr(char *s)` renvoyant `1` si la chaîne d'adresse `s` est à tranches incrémentales, et `0` sinon. La chaîne ne devra être parcourue qu'une fois au plus.

Exercice 4

On numérote les tranches des chaînes à partir de 0. Deux chaines seront dites *à tranches similaires* si et seulement si :

- elles ont le même nombre de tranches,
- les tranches de même numéro dans les deux chaînes sont formées du même caractère – elles peuvent, bien sûr, être de longueurs différentes.

Par exemple, les chaînes `"aaabccccd"` et `"abbbbcccdddd"` sont à tranches similaires. Ecrire `int tranches_similaires(char *s, char *t)` renvoyant `1` si les chaînes d'adresses `s` et `t` sont à tranches similaires, et `0` sinon. Chaque chaîne ne devra être parcourue qu'une fois au plus.

Exercice 5

Etant donnés une chaîne s et un entier $n > 0$, appelons *forme régulière de rang* n de s la chaîne t vérifiant les deux deux propriétés suivantes :

- s et t sont à tranches similaires (*cf.* l'exercice précédent),
- chaque tranche de t est de longueur n.

Par exemple, la forme régulière de rang 1 de `"aaabccccd"` est `"abcd"` ; sa forme régulière de rang 2 est `"aabbccdd"`; sa forme régulière de rang 3 est `"aaabbbcccddd"`, etc.

A l'aide de la fonction `nbr_tranches` de l'exercice 1 et de `malloc`, écrire une fonction `char *forme_reguliere(int n, char *s)`. Cette fonction doit renvoyer l'adresse d'une chaîne allouée par `malloc`, égale à la forme régulière de rang `n` de la chaîne d'adresse `s`.

La chaîne d'adresse `s` ne devra être parcourue que deux fois en tout : une première fois, par un appel la fonction `nbr_tranches` ; une seconde fois, pour construire la chaîne résultat.

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 fonctions ci-dessous doivent être écrites par récurrence pure : aucun `while`, aucun `for`. Chaque fonction ne peut appeler qu'elle-même et `free`. N'oubliez pas le ou les cas de base, et vérifiez soigneusement l'ordre des opérations et la syntaxe des appels récursifs.

Exercice 6

Ecrire `struct cell *suppr_der(struct cell *pc)`. Si la liste `pc` et vide, cette fonction renvoie la liste vide. Sinon, elle doit renvoyer la liste obtenue en supprimant la *dernière* cellule de `pc`.

Exercice 7

Ecrire `struct cell *suppr_av_der(struct cell *pc)`. Si la liste `pc` contient au plus une cellule, cette fonction se contente de renvoyer `pc`. Sinon, elle doit renvoyer la liste obtenue en supprimant l'*avant-dernière* cellule de `pc`.

III - Problème

Le but de cette dernière partie, un peu plus libre, est de montrer votre capacité à résoudre de manière concise un problème simple, mais dont l'implémentation peut être de qualité très variable. Votre code devra être le plus soigné et le moins redondant possible.

Le problème du mille-pattes Etant donnée une matrice d'entiers M de taille $H \times L$, on dira que cette matrice est un *mille-pattes* si et seulement si elle satisfait les deux conditions suivantes :

- M contient une et une seule fois chaque entier entre 0 et $(H \times L) - 1$.
- pour chaque entier n de M qui n'est pas le plus grand, le successeur de n est situé immédiatement au dessus, en dessous, à gauche ou à droite de n.

Par exemple, la matrice ci-dessous est un mille-pattes, comme l'indique la suite de déplacements indiquée par les flèches :

$$\left(\begin{array}{ccccccccc}
15 & \rightarrow & 16 & \rightarrow & 17 & \rightarrow & 18 & \rightarrow & 19 \\
\uparrow & & & & & & & & \\
14 & \leftarrow & 13 & & 0 & & 3 & \rightarrow & 4 \\
& & \uparrow & & \downarrow & & \uparrow & & \downarrow \\
11 & \rightarrow & 12 & & 1 & \rightarrow & 2 & & 5 \\
\uparrow & & & & & & & & \downarrow \\
10 & \leftarrow & 9 & \leftarrow & 8 & \leftarrow & 7 & \leftarrow & 6
\end{array}\right)$$

Question 1 Donnez, en langage naturel et en quelques lignes, une méthode simple et efficace permettant de vérifier si une matrice est un mille-pattes.

Question 2 Précisez les types de données choisis Vous pouvez introduire de nouveaux types (des structures, des énumérations) en en donnant la déclaration. Le choix de la représentation des matrices est libre : vous pouvez choisir des matrices de taille constante, ou dynamiquement ajustable (inutile d'écrire dans ce cas d'écrire les fonctions d'allocation et de libération associées).

Question 3 Donnez une implémentation de votre méthode, idéalement en découpant de manière rationnelle le traitement en plusieurs fonctions (dont, bien sûr, une fonction principale renvoyant `1` si une matrice donnée en argument est un mille-pattes, et `0` sinon).

Casse-tête final Si vous avez traité *tout* ce qui précède, trouvez et implémentez une méthode permettant de générer aléatoirement une matrice mille-pattes – toutes les matrices mille-pattes doivent avoir les même chances d'être générées. Il faut utiliser la récurrence. Rappel : l'expression `(rand() % N)` sévalue en un nombre aléatoire compris entre `0` et `N - 1` inclus.