Programmation en C

Examen du 6/01/2011 - 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.

Matrices

Exercice 1

Une matrice d'entiers sera dite *en escalier* si :

- la dernière valeur non nulle de chaque ligne (en parcourant cette ligne de gauche à droite) n'est pas plus à gauche que la dernière valeur non nulle de la ligne précédente,
- la première valeur non nulle de chaque colonne (en parcourant cette colonne du haut vers le bas) n'est pas plus haute que la première valeur non nulle de la colonne précédente.

Par exemple, la matrice suivante est en escalier:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 7 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 5 & 0 & 5 & 3 & 0 & 0 \\ 0 & 4 & 0 & 4 & 3 & 0 \end{pmatrix}$$

Ecrire `int en_escalier(int m[M][N])` où `M`, `N` sont deux constantes quelconque définies en début de programme, renvoyant `1` si `m` est une matrice en escalier et `0` sinon.

Le choix de la méthode est libre, mais votre solution devra être à la fois simple et efficace. Le contenu de la matrice ne devra être lu qu'au plus une fois, et la fonction devra renvoyer sa réponse le plus rapidement possible.

Chaînes

Exercice 2

Ecrire une fonction `char *copier(char *s, int pos, int nbr)`. Cette fonction doit allouer, construire, puis renvoyer l'adresse de départ d'une nouvelle chaîne contenant tous les caractères de `s` dont les positions sont comprises entre `pos` et `pos + (nbr - 1)` inclus, dans l'ordre où ils apparaissent dans `s`.

Noter que ces conventions autorisent `pos` à être négatif ou au delà de la dernière position de `s`, ou encore, `nbr` à être nul ou négatif. Parmi toutes les positions spécifiées par `pos` et `nbr`, on ignorera simplement toutes celles qui ne sont pas des positions dans `s`. Lorsque par exemple `s` contient `"abcde"` :

- `copier(*s, 2, 2)` renvoie "cd", (positions 2, 3)

- `copier(*s, 2, -3)` renvoie `"abc"` (2, 1, 0)
- `copier(*s, 2, 5)` renvoie `"cde"` (2, 3, 4, puis 5, 6 ignorées)
- `copier(*s, -2, 4)` renvoie `"ab"` (-2, -1 ignorées, puis 0, 1)
- `copier(*s, 2, 0)` renvoie `""` (et non un pointeur nul)

Ecrire ensuite une fonction `char *couper(char *s, int pos, int nbr)`. Cette fonction doit allouer, construire, puis renvoyer l'adresse de départ d'une nouvelle chaîne contenant tous les caractères de `s` dans l'ordre où ils apparaissent dans `s`, *sauf* ceux dont les positions sont comprises entre `pos` et `pos + (nbr - 1)` inclus.

Ecrire enfin une fonction `char *coller(char *s, int pos, char *t)` allouant, construisant, puis renvoyant l'adresse de départ d'une nouvelle chaîne contenant successivement : tous les caractères de `s` de position au plus `pos`, puis, tous les caractères de `t`, puis, tous les caractères de `s` de position au moins `pos`.

Exercice 3

Une chaîne sera dite *progressive par la droite* s'il elle est formée de la suite de tous les préfixes successifs d'un même mot. Par exemple, `aababcabcdabcde` est progressive par la droite, formée de la suite des préfixes de `abcde`. Ecrire `int progr_dr(char *s)` renvoyant `1` si `s` est progressive par la droite, `0` sinon.

Symétriquement, une chaîne sera dite *progressive par la gauche* s'il elle est formée de la suite de tous les suffixes successifs d'un même mot. Par exemple, `edecdebcdeabcde` est progressive par la gauche, formée de la suite des suffixes de `abcde`. Ecrire `int progr_ga(char *s)` renvoyant `1` si `s` est progressive par la gauche, `0` sinon.

Remarque Dans les deux cas, inutile de chercher en fin de chaîne le mot ayant servi à construire `s`. Chaque test est en fait plus simple que sa définition.

Listes chaînées

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, `malloc`. Attention aux cas limites.

Exercice 4

Une liste est *incrémentale* si chaque clef après la première est égale à celle qui la précède plus un. Par exemple, la liste contenant successivement $3, 4, 5, 6$ est incrémentale, de plus petite valeur 3.

Symétriquement, une liste est *décrémentale* si chaque clef après la première est égale à la précédente moins un. Par exemple, la liste contenant successivement $6, 5, 4, 3$ est décrémentale, de plus petite valeur 3.

Noter que par convention, une liste vide ou réduite à un seul élément est à la fois incrémentale et décrémentale.

Question 1 Ecrire les fonctions suivantes:

1. `struct cell *incrementale(int n, int min)`. Cette fonction doit allouer et construire une liste incrémentale de longueur `n` et de plus petite valeur `min` si `n` est non nul, puis renvoyer l'adresse de sa première cellule.

2. `struct cell *decrementale(int n, int min)`. Cette fonction doit allouer et construire une liste décrémentale de longueur `n` et de plus petite valeur `min` si `n` est non nul, puis renvoyer l'adresse de sa première cellule.

Question 2 On suppose définies en début de programme quatre constantes `ID`, `I`, `D`, `N` de valeurs quelconques, mais distinctes. En utilisant le moins de `if` possible, écrire `int type_increment(struct cell *pc)` renvoyant :

- `ID` si la liste `pc` est à la fois incrémentale et décrémentale,
- `I` si elle est seulement incrémentale,
- `D` si elle est seulement décrémentale,
- `N` si elle n'est ni incrémentale, ni décrémentale.