Programmation en C
Examen du 4/01/2006

Bien qu'inutiles, les notes de cours sont autorisées, pas les livres. N'oubliez pas de commenter votre code.

Exercice

Etant donnée une matrice d'entiers, on appelle *partie utile* de cette matrice le plus petit rectangle contenant tous les éléments non nuls de cette matrice. Par exemple, dans la matrice `m` ci-dessous,

$$\begin{array}{cc} & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} & \boxed{\begin{array}{ccccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 17 & 5 & 0 \\ 0 & 42 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}} \end{array}$$

la partie utile est le rectangle dont le coin supérieur gauche est `m[2][1]` - ligne n^o `2`, colonne n^o `1` - de largeur `3`, de hauteur `2`. Ecrire une fonction :

```
struct rectangle partie_utile(int m[TAILLE][TAILLE])
```

renvoyant une description de la partie utile de `m`, sous la forme d'une structure dont le type sera déclaré par :

```
struct rectangle {
  int ligne;
  int colonne;
  int largeur ;
  int hauteur;
};
```

Les champs `ligne` et `colonne` devront contenir les coordonnées du coin supérieur gauche du rectangle, les champs `largeur` et `hauteur` devront contenir sa largeur et sa hauteur. La constante `TAILLE` sera supposée déclarée en début de programme, et de valeur quelconque. Dans le cas où la `m` ne contient que des `0`, tous les champs du résultat devront valoir `0`.

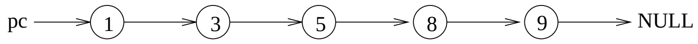
Remarque. Cette fonction peut être écrite en ne lisant qu'une et une seule fois chaque élément de `m`, quel que soit son contenu. Attention à ne pas confondre lignes et colonnes.

Exercice

On considère le type usuel permettant de représenter en mémoire des listes chaînées étiquetées par des entiers :

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

Comme d'habitude, une liste sera représentée par un pointeur vers sa première cellule, et la liste vide par un pointeur nul. Par exemple, voici la représentation en mémoire de la liste contenant successivement `1`, `3`, `5`, `8` et `9` :



Ecrire par récurrence les fonctions suivantes. L'utilisation de la récurrence est impérative.

- `int egales(struct cell *pc1, struct cell *pc2)`

  renvoyant `1` si la suite des clefs de `pc1` est exactement égale à la suite des clefs de `pc2`, et `0` sinon.

- `int prefixe(struct cell *pc1, struct cell *pc2)`

  renvoyant `1` si la suite des clefs de `pc1` est un préfixe de la suite des clefs de `pc2`, et `0` sinon. Par exemple, la liste contenant successivement `1`, `3` et `5` est préfixe de la liste ci-dessus.

- `struct cell *copier(struct cell *pc)`

  renvoyant une copie de `pc`, *i.e* une liste contenant la même suite de clefs, mais dont chaque cellule est une nouvelle cellule allouée par `malloc`.

Problème

Le problème de la *compression* d'un texte a de nombreuses solutions, mais toutes partent d'une observation simple : un texte en langage naturel est écrit avec un nombre limité de mots, et certains mots apparaissent en général plusieurs fois dans un même texte.

Une méthode de compression élémentaire consiste à construire une version simplifiée du texte dans laquelle on élimine toutes les répétions de mots. Cette version simplifiée est appelée le *lexique* du texte.

On peut alors décrire le contenu du texte par une simple suite de positions dans le lexique : "*le mot à la position* `0`*, puis le mot à la position* `3`*, ..., à nouveau le mot à la position* `0`*, etc.*", représentable sous forme d'un tableau d'entiers. S'il y a peu de mots, si les mots sont assez longs et si le texte contient beaucoup de répétitions, cette méthode peut même être assez efficace.

Conventions

On convient du fait que les mots du texte à compresser ainsi que ceux de son lexique sont séparés par des espaces simples, sans espaces initiaux ou finaux. Si par exemple le texte initial est `"to be or not to be"`, son lexique est :

0	1	2	3	4	5	6	7	8	9	10	11	12
't'	'o'	' '	'b'	'e'	' '	'o'	'r'	' '	'n'	'o'	't'	'\0'

et sa forme compressée est `[0, 3, 6, 9, 0, 3, -1]` La dernière valeur `-1` ne correspond à aucune position dans le lexique, mais signale simplement dans le tableau la fin de la suite de positions.

Question 1

Ecrire les fonctions suivantes :

1. `void afficher_lexique(char *lexique)`

   cette fonction doit afficher chaque mot de `lexique`, en effectuant un retour-chariot après chaque mot.

2. `void afficher_mot(int pos, char *lexique)`

   cette fonction suppose que `pos` est la position du début d'un mot dans `lexique`. Elle doit afficher ce mot.

3. `void afficher_texte(int *comp, char *lexique)`

   cette fonction suppose que ses arguments sont un texte compressé et son lexique : `comp` est une suite de positions de mots dans `lexique`, suivie d'un `-1`. En se servant de la fonction précédente, elle doit réafficher le texte initial - sans oublier les espaces.

Question 2

Ecrire les fonctions suivantes :

1. `int taille_lexique(char *lexique)`

   cette fonction doit renvoyer le nombre total de mots du lexique (*pas* son nombre de caractères).

2. `int taille_mot(int pos, char *lexique)`

   cette fonction suppose que `pos` est la position du début d'un mot dans `lexique`. Elle doit renvoyer la taille de ce mot.

3. `int pos_suivante(int pos, char *lexique)`

   cette fonction suppose que `pos` est la position du début d'un mot dans `lexique`. Elle doit renvoyer la position de début du mot suivant s'il existe, et `-1` sinon. Elle peut se servir de la précédente.

Question 3

Ecrire les fonctions suivantes. Vous pouvez (et même, raisonnablement, devez) vous servir de la fonction `taille_mot` de la question 2, et de `malloc`.

Chaque fonction suppose que ses arguments sont un texte compressé et son lexique : `comp` est une suite de positions de mots dans `lexique`, suivie d'un `-1`.

- `void taille_texte(int *comp, char *lexique)`

  Cette fonction doit calculer la taille totale du texte avant sa compression.

- `char *decompresser(int *comp, char *lexique)`

  à l'aide de la fonction précédente, cette fonction doit allouer une zone mémoire exactement assez grande pour contenir le texte dont `comp` est la compression.

  Elle doit ensuite reconstruire ce texte initial dans la zone allouée, et renvoyer l'adresse de cette zone.