TP9 : Programmation dynamique et fonction de hashage

Algorithmique et projet de programmation — M1 mathématiques

29 novembre 2016

1 Plus long sous-mot commum

Rappelons qu'un alphabet Σ est juste un ensemble fini dont on appelle les éléments *lettres*. Un mot w sur un alphabet Σ est une séquence finie $(a_1, \ldots, a_n)$ avec tous les $a_i \in \Sigma$. On ommettra les parenthèses et les virgules, non nécessaires, et on notre $w = a_1 \ldots a_n$. La longueur d'un mot est simplement le nombre de lettres qui le composent. L'ensemble de tous les mots sur un alphabet Σ est noté Σ^*.

Un sous-mot d'un mot $w = a_1 \ldots a_n \in \Sigma^*$ est un mot $u \in \Sigma^*$ tel qu'il existe des indices $1 \leq i_1 < \cdots < i_k \leq n$ tels que $u = a_{i_1} \ldots a_{i_k}$. En français, c'est dire qu'en supprimant certaines lettres de w, on tombe sur u. On s'intéresse ici à développer un algorithme renvoyant un sous-mot le plus grand possible commun à deux mots $w, w' \in \Sigma^*$ donnés.

En écrivant $w = a_1 \ldots a_n$ et $w' = b_1 \ldots b_m$ (avec $a_i, b_j \in \Sigma$), on peut caractériser les sous-mots communs de taille maximale comme suit :

- soit n ou m est nul et le seul sous-mot commun est le séquence vide ε de longueur 0,
- soit $a_n = b_m = x$, et tout sous-mot commun maximal u de $a_1 \ldots a_{n-1}$ et $b_1 \ldots b_{m-1}$ donne un sous-mot commun maximal ux de w et w',
- soit $a_n \neq b_m$ et un sous-mot commun maximal de w et w' est un sous-mot maximal commun de w et $b_1 \ldots b_{m-1}$ ou de $a_1 \ldots a_{n-1}$ et w'.

Exercice 1. En déduire un algorithme récursif donnant la longueur d'un sous-mot maximal de deux mots donnés w et w'.

Cet algorithme est-il efficace ? Pourquoi ?

Pour éviter de répéter des calculs déjà effectués, on va utiliser une technique de *programmation dynamique* appelée *memoisation*. Elle consiste simplement à stocker les valeurs déjà calculées d'une fonction, afin qu'à un appel ultérieure, on puisse retourner directement la valeur au lieu de la recalculer bêtement.

Exercice 2. Écrire une fonction :

```
int memoized_lcs (char *v, char *w, int i, int j, int **computed);
```

prenant en paramètres deux mots $v = v_1 \dots v_n$ et $w = w_1 \dots w_m$, deux indices i et j, et une matrice d'entiers `computed` et renvoyant la longueur d'un plus long sous-mot commun (*Longest Common Subsequence* en anglais) de $v_1 \dots v_i$ et $w_1 \dots w_j$. On suppose que la matrice `computed` est de taille $(n+1) \times (m+1)$ (avec n la taille de v et m celle de w) et contient en case $(i\,,j)$

- soit la longueur, déjà calculée, d'un plus long sous-mot commun à $v_1 \dots v_i$ et $w_1 \dots w_j$,
- soit -1 si la valeur n'a pas encore été calculée.

En déduire une fonction efficace :

```
int lcs (char *v, char *w);
```

renvoyant la longueur d'un plus long sous-mot commun à v et w.

Exercice 3. Améliorer les fonctions précédentes pour retourner effectivement un sous-mot maximal commun de v et w, et non plus seulement sa longueur.

2 Table de hashage

Les tables de hashage sont le pendant informatique des partitions d'un ensemble. Partitionner un ensemble (fini) S de taille $|S|$ en $m \in \mathbb{N}$ sous-ensembles est équivalent à se donner une fonction $h : S \to \{0\,, \dots, m-1\}$: en effet, une telle fonction étiquette chacun des éléments de S avec le numéro i du sous-ensemble $h^{-1}(i)$ de la partition le contenant.

En terme du langage C, cela veut dire que si l'on dispose d'une fonction de la forme `int h(typ x)` à valeurs dans $\{0\,, \dots, m-1\}$ pour un certain $m \in \mathbb{N}$, alors on peut représenter un ensemble fini S d'éléments de type `typ` comme un tableau de taille m où la case indexée par `i` contient les éléments `x` de S tels que `h(x)` retourne `i`. Encore faut-il trouver un moyen de "faire rentrer" tous ces éléments dans une même case... On peut utiliser toute les structures de données que vous avez pu voir, dépendamment de l'application.

Pour ce TP, par souci de simplicité, on choisit d'utiliser des listes chaînées, c'est-à-dire que chaque case de notre table de hashage contiendra un pointeur vers le premier noeud d'une liste chaînée. Enfin, pour fixer les idées, le type abstrait `typ` sera ici le type des couples d'entiers :

```
struct coord_s {
  int x,y;
} coord_t;
```

Exercice 4 (au tableau)**.** Écrire des fichiers `coord.c` et `list.c` (et leurs headers) permettant de manipuler les couples et les listes chaînées de couples.

Dans un fichier `hash.c`, on va implémenter la structure C correspondant aux tables de hashage d'éléments de type `coord_t` :

```
struct hash_s {
  int size; // size of table
  list_t *table; // actual hash table
};
```

Ici, on veut cacher les détails de l'implémentation à l'utilisateur de notre bibliothèque `hash.c`. On va donc simplement lui donner accès à la structure via un pointeur dans `hash.h` :

```
#ifndef HASH_H
#define HASH_H

typedef struct hash_s* hash_t;

// other function declarations

#endif
```

Rappelez-vous qu'un tel `typedef`, sans definition explicite de la structure dans le header, doit nécessairement porter sur un pointeur. Un simple

```
typedef struct hash_s hash_t;
```

est prohibé : la déclaration d'une variable de type `hash_t` par l'utilisateur n'ayant accès qu'au header serait alors un vrai casse-tête pour C, qui, ne sachant pas combien de place allouer sur la pile d'exécution, préfère échouer à la compilation (erreur de type `incomplete type`). L'allocation d'un pointeur en revanche ne pose aucun problème à C : quelque soit l'objet vers lequel pointe celui-ci, il n'y a besoin que d'une case mémoire pouvant contenir une adresse de la RAM !

Exercice 5. Proposer une fonction mathématique

$$h : \mathbb{N} \times \mathbb{N} \to \{0, \ldots, m-1\}$$

qui soit une bonne fonction de hashage. L'implémenter en C.

Exercice 6. Ecrire les fonctions suivantes :

```
// create a (chained) hash table of given size
hash_t hash_init (int size);

// insert key inside htab, with value val
hash_t hash_ins (coord_t key, int val, hash_t htab);

// remove key from htab
hash_t hash_rem (coord_t key, hast_t htab);

// search for key in htab, returns 1 if found, 0 otherwise
char hash_mem (coord_t key, hast_t htab);

// return value contained at key in htab
// Careful: key should be member of htab
int hash_getval (coord_t key, hast_t htab);
```

3 Pour le 6 décembre 2016

Exercice 7.

(i) Modifier les fonctions du premier exercice afin d'afficher la matrice `computed` après l'appel à `memoized_lcs` sur la longueur de v et w. Que remarquer ? (Indice : utiliser de grandes chaînes de caractères.)

(ii) Utiliser la structure de table de hashage mise au point dans l'exercice 2 pour palier à ce problème.