Université Savoie Mont Blanc, UFR SceM
INFO504, L3 INFO

October 24, 2023

Problème 1, Analyse de textes

(Tableaux mono-dimensionnels, chaînes de caractères, fonctions, structures de données, pointeurs, passage par adresse, hachage)

On se donne un texte quelconque:

```
char texte[] = "Le petit poucet

Il était une fois un bûcheron et une bûcheronne qui avaient sept enfants, tous garçons; l'aîné n'
    avait que dix ans, et le plus jeune n'en avait que sept.

On s'étonnera que le bûcheron ait eu tant d'enfants en si peu de temps ; mais c'est que sa femme
    allait vite en besogne, et n'en avait pas moins de deux à la fois. ...";
```

On voudrait analyser le texte, par exemple le découper en mots, faire des statistiques sur la fréquence des lettres et des mots, savoir si un mot est utilisé, etc. Cela permet par exemple d'indexer le texte pour des recherches futures, ou de comparer deux textes pour savoir s'il y a eu plagiat.

Exemple : (Longueur d'une chaîne de caractères) On rappelle qu'une chaîne de caractères C est une suite de caractères consécutifs en mémoire. Le caractère de valeur 0 (ou '\0') termine la chaîne de caractères. Ainsi, la fonction calculant la longueur d'une chaîne peut s'écrire:

```
int strlen( char s[] )
{
  int n = 0;
  while ( s[ n ] != 0 ) n++;
  return n;
}
```

Question 1. Quelle est la taille du mot le plus long ? Quel est le nombre de mots ? (On suppose pour le moment que les mots sont séparés par des espaces.)

```
int size_longest_word( char t[] ); // t is the input string
int nb_words         ( char t[] ); // t is the input string
```

```
// @param t is the input string
//
// @return the length of the longest word in @a t (assuming ' ' is the
// sole separator).
int size_longest_word( char t[] )
{
  int cur_l = 0;
  int max_l = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
      if ( t[ i ] != ' ' ) cur_l += 1;
      else
        {
          max_l  = cur_l > max_l ? cur_l : max_l;
          cur_l = 0;
        }
    }
  return cur_l > max_l ? cur_l : max_l;
}
```

```
// @param t is the input string
//
// @return the number of words in @a t (assuming ' ' is the sole
// separator).
int nb_words( char t[] )
{
  bool word = false;
  int  nb_w = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
```

```
      if ( t[ i ] != ' ' ) word = true;
      else if ( word )
        {
          nb_w += 1;
          word  = false;
        }
    }
  return word ? nb_w + 1 : nb_w;
}
```

Question 2. Il y a beaucoup de séparateurs de mots (espaces, ponctuations, fin de ligne). Mêmes questions que précédemment avec des séparateurs.

```
int is_in_string      ( char c, char s[] ); // return true is c is a letter in s
int size_longest_word( char t[], char sep[] ); // sep contains the separators
int nb_words          ( char t[], char sep[] ); // sep contains the separators
```

Remarques : Si vous voulez utiliser le type `bool` plutôt que `int` pour désigner un booléen, il faut inclure `<stdbool.h>` (depuis C99).

```
// @param c any character
// @param s a C-string
// @return 'true' if and only if c belongs to string @a s.
bool is_in_string( char c, char s[] )
{
  for ( int i = 0; s[ i ] != 0; i++ )
    if ( s[ i ] == c ) return true;
  return false;
}
```

```
// @param t is the input string
// @param sep a C-string containing all the possible word separators
//
// @return the length of the longest word in @a t
int size_longest_word_sep( char t[], char sep[] )
{
  int cur_l = 0;
  int max_l = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
      if ( ! is_in_string( t[ i ], sep ) ) cur_l += 1;
      else
        {
          max_l  = cur_l > max_l ? cur_l : max_l;
          cur_l = 0;
        }
    }
  return cur_l > max_l ? cur_l : max_l;
}
```

```
// @param t is the input string
// @param sep a C-string containing all the possible word separators
//
// @return the number of words in @a t
int nb_words_sep          ( char t[], char sep[] )
{
  bool word = false;
  int  nb_w = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
      if ( ! is_in_string( t[ i ], sep ) ) word = true;
      else if ( word )
        {
```

```
            nb_w += 1;
            word  = false;
          }
      }
    return word ? nb_w + 1 : nb_w;
  }
```

Question 3. Savoir si un caractère est un séparateur prend un temps proportionnel au nombre de séparateurs. Pouvez-vous imaginer une structure de données et des fonctions qui permettent de déterminer plus rapidement si un caractère est un séparateur?

Pour éviter une boucle sur les caractères séparateurs possibles, l'idée est d'utiliser le fait qu'il n'y a que 256 caractères différents, et que les caractères sont codés comme des entiers. On va donc précalculer un tableau `table` à 256 valeurs booléennes, tel que si `c` est un caractère séparateur, alors `table[c] == true`.

On voit ensuite qu'il suffira d'utiliser cette table pour savoir si un caractère est un séparateur, ce qui ne demandera au processeur qu'un accès mémoire.

La seule petite difficulté est que le type `char` est signé et représente les entiers entre -128 et 127. Il faut donc "caster" l'entier signé en un entier non signé `unsigned char` dont les valeurs sont entre 0 et 255.

```
typedef unsigned char uint8;
typedef uint8 CharTable[ 256 ];

// @param[in] c any character
//
// @param[in] table a 256 entries table with, for each character,
// either 'true' for present or 'false' for absent
//
// @return 'true' iff @a c is marked 'true' in table @a table
bool is_true_in_char_table( char c, CharTable table )
{
  return table[ (unsigned char) c ];
}

// @param[out] table a 256 entries table such that 'table[ c ] ==
// true' iff @a c belongs to string @a s
//
// @param[in] s a C-string containing some characters
void make_char_table( CharTable table, char s[] )
{
  for ( int i = 0; i < 256; i++ )
    table[ i ] = false;
  for ( int i = 0; s[ i ] != 0; i++ )
    table[ (uint8) s[ i ] ] = true;
}

// @param[in] t is the input string
//
// @param[in] table a 256 entries table whose entries are 'true' for character separators
//
// @return the length of the longest word in @a t
int size_longest_word_table( char t[], CharTable table )
{
  int cur_l = 0;
  int max_l = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
      if ( ! is_true_in_char_table( t[ i ], table ) )
        cur_l += 1;
      else
        {
          max_l  = cur_l > max_l ? cur_l : max_l;
          cur_l = 0;
        }
    }
```

```
  return cur_l > max_l ? cur_l : max_l;
}

// @param t is the input string
//
// @param[in] table a 256 entries table whose entries are 'true' for character separators
//
// @return the number of words in @a t
int nb_words_table( char t[], CharTable table )
{
  bool word = false;
  int  nb_w = 0;
  for ( int i = 0; t[ i ] != 0; i++ )
    {
      if ( ! is_true_in_char_table( t[ i ], table ) )
        word = true;
      else if ( word )
        {
          nb_w += 1;
          word  = false;
        }
    }
  return word ? nb_w + 1 : nb_w;
}
```

On pourrait s'en servir ainsi:

```
int main( int argc, char* argv[] )
{
  char* fname = argc > 1 ? argv[ 1 ] : "le-petit-poucet.txt";
  char sep[]  = { ' ', ',', ';', '.', ':', '\"', '\'', '(', ')', '!', '?', '-',
                  '~', '`', 13, 10, 0 };
  /* Lecture du fichier */
  FILE* F     = fopen( fname, "r" );
  char* buffer= (char*) malloc( 1000000 );
  int   nread = fread( buffer, 1, 1000000, F );
  printf( "%s: %d bytes read.\n", fname, nread );
  buffer[ nread ] = 0; // termine la chaine par 0
  fclose( F );

  /* Exo 3 */
  CharTable sep_table;
  make_char_table( sep_table, sep );
  printf( "size longest word is %d (for generic sep_table)\n",
          size_longest_word_table( buffer, sep_table ) );
  printf( "number of words is %d (for generic sep_table)\n",
          nb_words_table( buffer, sep_table ) );
  return 0;
}
```

Question 4. Est-il facile de substituer un mot par un autre dans un texte (en C) ?

En fait non, sauf si le mot que l'on met à la place du mot choisi a la même longueur (ou nombre de caractères). Cela est dû au fait que les chaînes de caractères sont des octets consécutifs en mémoire. On ne peut insérer ou supprimer des caractères sans devoir décaler le reste du texte.

Question 5. Enlever une lettre dans une chaîne de caractères. Par exemple, on voudrait enlever tous les espaces ' ', les guillemets '"', etc.

```
int remove_letter( char c, char s[] ); // return the number of removed letters
                                       // the array stored at s is modified
```

```
// @param[in] c any character
// @param[inout] s a C-string from which all @a c characters are removed.
//
// @return the number of removed letters
int remove_letter( char c, char s[] )
{
  int i = 0; // tête de lecture
  int j = 0; // tête d'écriture
  for ( ; s[ i ] != 0; i++ )
    {
      if ( s[ i ] != c )
        s[ j++ ] = s[ i ];
    }
  s[ j ] = 0;
  return i - j;
}
```

Question 6. On veut savoir combien de fois chaque mot apparaît, connaître le mot le plus fréquent. Est-ce facile d'écrire une fonction qui réalise ces opérations ?

Identifier informellement des structures de données, fonctions intermédiaires.

On voit qu'il faut identifier des mots dans le texte, plutôt que de raisonner au niveau des caractères. Il faut donc avoir une structure de données pour représenter facilement un mot et lui associer des quantités (comme le nombre de fois où il apparaît dans un texte).

Il faut donc être capable de découper un texte en ses mots et ensuite avoir un dictionnaire qui répertorie les mots utilisés ainsi que le nombre de fois où il apparaît.

En algorithmique, on utiliserait volontiers un "trie", ou arbre préfixe. On fera plus simple ainsi avec un simple tableau de mots.

Question 7. Découper un texte mot à mot.

On se donnera une structure `Token` pour stocker le mot courant, puis on écrira cinq fonctions pour manipuler les `Token`. Elles auront pour prototypes:

```
// Initialise le token sur le premier mot du texte
void  Token_init    ( Token* t, char str[], char sep[] );
// Retourne le mot courant stocké dans le token
char* Token_valeur ( Token* t);
// Retourne vrai si on est à la fin du texte (mot courant vide).
bool  Token_fini    ( Token* t);
// Passe au mot suivant du texte (valide si !Token_fini(...))
void  Token_suivant( Token* t);
// Indique que l'on a fini de se servir du token.
void  Token_termine( Token* t);
```

Ainsi le programme suivant

```
int main()
{
  char s   [] = "Nous irons, tous, au    paradis";
  char sep[] = { ' ', '\n', ',', '.', ';', ':', '?', '!', '\'', '\"' };
  Token tok;
  Token_init( &tok, s, sep );
  for( ; ! Token_fini( &tok ); Token_suivant( &tok ) )
    printf( "%s|", Token_valeur( &tok ) );
  Token_termine( &tok );
}
```

affichera:

```
Nous|irons|tous|au|paradis|
```

Dans la correction ci-dessous nous avons choisi de stocker un pointeur vers les séparateurs, et de tester simplement si un caractère est un séparateur en le cherchant dans la liste des séparateurs possibles. Pour optimiser le code (notamment sur des textes longs), nous aurions pu utiliser la méthode par table de l'exercice 3. A ce moment, nous aurions stocké une table dans la structure `Token` et nous aurions appelé `make_char_table` dans `Token_init`, et `is_true_in_char_table` pour tester si un caractère est un séparateur.

```
/* token.h */
#ifndef __TOKEN_H__
#define __TOKEN_H__

#define SIZE_WORD 256

typedef struct {
  char* text;
  char* sep;
  char  word[ SIZE_WORD ];
} Token;

int    is_in_string ( char c, char* s );
void  Token_init    ( Token* t, char str[], char sep[] );
char* Token_valeur ( Token* t);
int    Token_fini    ( Token* t);
void  Token_suivant( Token* t);
void  Token_termine( Token* t);

#endif
```

```
/* token.c */
#include <stdlib.h>
#include "token.h"

int    is_in_string      ( char c, char* s )
{
  for ( ; *s ; s++ )
    if ( *s == c ) return 1;
  return 0;
}
```

```
void  Token_init   ( Token* t, char str[], char sep[] )
{
  t->text = str;
  t->sep  = sep;
  Token_suivant( t );
}

char* Token_valeur ( Token* t)
{
  return t->word;
}

int   Token_fini   ( Token* t)
{
  return t->word[ 0 ] == 0;
}

void  Token_suivant( Token* t)
{
  while ( is_in_string( t->text[ 0 ], t->sep ) ) t->text++;
  int i = 0;
  while ( t->text[ 0 ] && ! is_in_string( t->text[ 0 ], t->sep ) )
    t->word[ i++ ] = *t->text++;
  t->word[ i ] = 0;
  while ( is_in_string( t->text[ 0 ], t->sep ) ) t->text++;
}

void  Token_termine( Token* t)
{
  t->text = NULL;
}
```

Question 8. Ecrivez maintenant une fonction qui détermine si un mot appartient à un texte.

```
// retourne 0 si s1 == s2, négatif si s1 < s2, positif si s2 < s1
int mystrcmp( const char* s1, const char* s2 );
// retourne vrai si w est un préfixe de s
bool prefix( const char* w, const char* s );
// retourne vrai si w est une sous-chaîne de s
bool substring( const char* w, const char* s );
```

Remarques : Est-ce que votre fonction est efficace ? Le temps de calcul dépend-il de la taille du mot, de la taille du texte ?

```
int mystrcmp( const char* s1, const char* s2 )
{
  while ( ( *s1 ) && ( *s2 ) && ( *s1 == *s2 ) )
    s1++, s2++;
  return ( (int) *s1 ) - ( (int) *s2 );
}

bool prefix( const char* w, const char* s )
{
  while ( ( *w ) && ( *s ) && ( *w == *s ) )
    w++, s++;
  return ( *w == 0 ) && ( ( *s == 0 ) || ( *w <= *s ) );
}

bool substring( const char* w, const char* s )
{
  while ( *s != 0 )
    if ( prefix( w, s ) ) return true;
    else s += 1;
  return false;
}
```

Le temps d'exécution dépend d'une double boucle, et est la somme des temps d'exécution de `prefix` à chacune des positions possibles dans le texte. Chercher un mot dans un texte peut prendre un temps proportionnel à la taille du mot le plus court multiplié par la taille du mot le plus long.

Par ailleurs, comparer l'égalité de deux mots peut prendre un temps proportionnel à la taille du mot le plus court. On voit donc qu'on a intérêt à découper un texte en mots (pour ne comparer que l'égalité de mots). On va voir ci-dessous qu'on peut aller encore plus vite pour tester l'égalité de mots.

Question 9. On propose les fonctions suivantes (dites de hachage)

```
int hash1( char* str ) // hachage simple
{
  int v = 0;
  while ( *str != 0 ) { v += *str++; }
  return v;
}
int hash2( char* str ) // hachage un peu plus évolué
{
  static int coef[] = { 7, 251, 89437, 17, 765411, 5, 1379, 23453 };
  int v = 0, i = 0;
  for ( int i = 0; str[ i ] != 0; ++i )
      v += str[ i ] * coef[ i & 7 ];
  return v;
}
```

Trouvez deux mots différents pour lesquels `hash1` donne la même valeur. Même question pour `hash2` ? On va s'en servir pour trouver les mots.

Pour `hash1`, on voit que, par exemple, une permutation du mot donne la même valeur, e.g. `hash1( "odil" ) == hash1( "lido" )`. Pour `hash2`, il y a clairement des collisions mais elles ne sont pas faciles à trouver, et sans doute peu probables. Ces fonctions donnent donc une première caractérisation du mot, que l'on peut vérifier avant de tester la vraie égalité.

Notez que ce principe de hashage est utilisé routinement en cryptographie (les empreintes MD5 ou SHA), ou dans les structures de données représentant des ensembles ou tableaux associatifs (tables de hachage).

Question 10. Proposez une structure de données pour stocker les mots d'un texte, le nb de fois où il apparait, leurs hachés.

```
struct SWord { char word[ 20 ]; int length; int h; int nb; };
typedef struct SWord Word;
struct SDico {  ...  };
typedef struct SDico Dico;
void  dico_init  ( Dico* ptrD );
void  dico_insert( Dico* ptrD, char* w );
Word* dico_search( Dico* ptrD, char* word ); // retourne null si non trouvé
```

Remarques : Pour le moment on suppose que le nombre maximum de mots est borné par une constante `MAX_NB_MOTS` et que la taille d'un mot est de 20 max.

L'utilisation du dictionnaire pourrait ressembler à ci-dessous:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "token.h"

int main( int argc, char* argv[] )
{
  char* fname = argc > 1 ? argv[ 1 ] : "le-petit-poucet.txt";
  char sep[]  = { ' ', ',', ';', '.', ':', '\"', '\'', '(', ')', '!', '?', '-',
                  '~', '`', 13, 10, 0 };
  /* Lecture du fichier */
  FILE* F     = fopen( fname, "r" );
  char* buffer= (char*) malloc( 1000000 );
  int   nread = fread( buffer, 1, 1000000, F );
  printf( "%s: %d bytes read.\n", fname, nread );
  buffer[ nread ] = 0;
```

```
fclose( F );

/* Lecture mot a mot des caracteres. */
Token T;
Dico  D;
dico_init( &D );
for ( Token_init( &T, buffer, sep ); ! Token_fini( &T ); Token_suivant( &T ) )
  {
    char* word = Token_valeur( &T );
    dico_insert( &D, word );
  }
...
```

```
/* dico.h */
struct SDico {
  Word  words[ MAX_NB_MOTS ];
  int   nb_words;
};
typedef struct SDico Dico;
void  dico_init   ( Dico* ptrD );
void  dico_insert ( Dico* ptrD, char* w );
Word* dico_search ( Dico* ptrD, char* w ); // retourne null si non trouvé

/* dico.c */
void  dico_init   ( Dico* ptrD )
{
  ptrD->nb_words = 0;
}

void  dico_insert ( Dico* ptrD, char* w )
{
  Word* cur = dico_search ( ptrD, w );
  if ( cur != NULL )
    cur->nb += 1;
  else
    {
      int   j     = ptrD->nb_words;
      Word* cur   = & ptrD->words[ j ];
      strcpy( cur->word, w );
      cur->length = strlen( w );
      cur->h      = hash1( w );
      cur->nb     = 1;
      ptrD->nb_words += 1;
    }
}

Word* dico_search ( Dico* ptrD, char* w )
{
  int lw = strlen( w );
  int hw = hash1 ( w );
  for ( int i = 0; i < ptrD->nb_words; i++ )
    {
      Word* cur = ptrD->words + i;
      if ( ( cur->length == lw ) && ( cur->h == hw )
           && strcmp( cur->word, w ) == 0 )
        return cur;
    }
  return NULL;
}
```

```
/* Exo 10 */
/* Création du dictionnaire. */
Dico D;
dico_init( &D );
for ( Token_init( &T, buffer, sep ); ! Token_fini( &T ); Token_suivant( &T ) )
  {
    char * word = Token_valeur( &T );
    dico_insert( &D, word );
  }
Token_termine( &T );
dico_display( &D );
```

Question 11. Proposez maintenant une approche pour rendre le temps de recherche d'un mot logarithmique en le nombre de mots, et (quasi indépendant de la taille des mots).

L'idée la plus naturelle est de trier les mots dans le dictionnaire pour pouvoir faire une recherche dichotomique. Attention, si on les mets dans l'ordre alphabétique, le tri aura un coût relativement élevé de l'ordre de $k*n\log n$ où n est le nombre de mots. Une façon plus astucieuse est de trier en fonction du haché de chaque mot. La recherche sera en fonction du haché. En cas d'égalité de hachés, il faudra regarder les mots consécutifs pour chercher le bon.

Dans tous les cas, cette structure n'est efficace que si le dictionnaire est complet. Dès qu'on rajoute des mots, il faut retrier tout.

En général, on utilisera donc d'autres structures de données plus souples pour représenter les dictionnaires, typiquement des tableaux associatifs (basés arbre ou table de hachage) ou des tries (arbres préfixes).

```
/* dico.h */
#include <stdlib.h>

void  dico_sort    ( Dico* ptrD );
Word* dico_fast_search ( Dico* ptrD, char* w );
```

```
/* dico.c */
int word_compare( const void* p1, const void* p2 )
{
  Word* w1 = (Word *) p1;
  Word* w2 = (Word *) p2;
  int    h1 = w1->h;
  int    h2 = w2->h;
  if ( h1 != h2 ) return h1 - h2;
  return strcmp( w1->word, w2->word );
}

void  dico_sort( Dico* ptrD )
{
  qsort( ptrD->words, ptrD->nb_words, sizeof( Word ), word_compare );
}

Word* dico_fast_search ( Dico* ptrD, char* w )
{
  int  lw    = strlen( w );
  int  hw    = hash1 ( w );
  int  i     = 0;
  int  j     = ptrD->nb_words - 1;
  Word nw    = { "X", lw, hw, 0};
  strcpy( nw.word, w );
  bool found = false;
  while ( ! found && ( i < j ) )
    {
      int m = (i+j) / 2;
      int order = word_compare( &nw, &ptrD->words[ m ] );
      if ( order < 0 )       j = m;
      else if ( order > 0 ) i = m + 1;
      else { i = m; found = true; }
    }
  return found ? ptrD->words + i : NULL;
}
```

```
/* Dans votre main */
/* Exo 11 */
/* Tri du dictionnaire. */
dico_sort    ( &D );
dico_display( &D );

ASSERT( dico_fast_search ( &D, "de" )->nb == 147 );
ASSERT( dico_fast_search ( &D, "brille" )->nb == 1 );
ASSERT( dico_fast_search ( &D, "parfaitement" )->nb == 2 );
ASSERT( dico_fast_search ( &D, "douleur" )->nb == 3 );
```