TP #26 — Recherche de facteurs

Exercice 1. *Algorithme « de Boyer-Moore »*

Le but de cet exercice est simplement d'implémenter l'algorithme « de Boyer-Moore » en OCaml, dans la version vue en cours.

On commence par implémenter une recherche de facteur naïve, sur des mots représentés par des `string`s.

1. Écrivez une fonction OCaml :
   `is_prefix_off : int -> string -> string -> bool`
   telle que `is_prefix_off off p s` s'évalue à `true` si `p` est un préfixe de `s` *avec décalage* `off` (c'est à dire que `s.[off] = p.[0]` etc.)
2. Testez dans des cas variés. (Faites éventuellement attention au fait que certains caractères ont une taille (d'encodage) supérieure à 1.)
3. Déduisez de ce qui précède une fonction :
   `is_prefix : string -> string -> bool`
   qui décide si son premier argument est préfixe de son second.
4. Testez.
5. Écrivez maintenant une fonction OCaml :
   `is_fact : string -> string -> int option`
   qui s'évalue à `None` si `f` n'est pas facteur de `s`, et sinon `Some i` où `i` est le décalage de la première occurrence de `f` dans `s` (c'est à dire que l'on a `f.[0] = s.[i]` etc.)
   Essayez de faire en sorte que cette fonction (bien que naïve) ne fasse pas de calculs inutiles outre mesure. (Par exemple en utilisant une exception pour interrompre une éventuelle boucle `for`.)
6. Testez.

On va maintenant implémenter l'algorithme « de Boyer-Moore »

7. Écrivez une fonction OCaml :
   `build_table : string -> (char, int) Hashtbl.t`
   telle que `build_table s` s'évalue en une table de hachage qui pour chaque `char` présent dans `s` y associe l'indice de sa dernière occurrence dans `s`.
   Vous pouvez par exemple essayer d'utiliser `String.iteri`, même si cette dernière fonction n'est pas au programme.
8. Écrivez une fonction OCaml :
   `get_largest_index : (char, int) Hasthbl.t -> char -> int`
   qui pour un premier argument construit par `build_table s` s'évalue en le plus grand index où `c` apparaît dans `s`, ou `-1` s'il n'y apparaît pas.

9. Testez.
10. Écrivez une fonction OCaml :

    `is_fact_bm : string -> string -> int option`

    de mêmes spécifications que `is_fact` mais qui utilise l'algorithme « de Boyer-Moore ».

    On rappelle brièvement que pour tester si `f` est facteur de `s`, celui-ci teste « de gauche à droite » (jusqu'à) tous les décalages pour déterminer si `f` apparaît dans `s` avec ce décalage `j`, en comparant les lettres « de droite à gauche », et en cas de désaccord à lettre d'indice `i` le prochain décalage à être testé est le maximum de `i - k` et `j + 1`, où `k` est renvoyé par un appel adapté à `get_largest_index`.

    Vous prendrez garde à justifier la validité de tous vos accès aux éléments des chaînes de caractères.

    L'utilisation d'exceptions pour la gestion du flot de contrôle peut à nouveau être utile.
11. Testez.

Exercice 2. *Algorithme « de Rabin-Karp »*

Le but de cet exercice est d'implémenter en C l'algorithme « de Rabin-Karp » dans la version vue en cours, *ainsi que de pratiquer la* **compilation séparée** : votre programme devra être écrit dans deux fichiers `.c`, l'un pour l'implémentation de la fonction de hachage incrémentale et l'autre pour l'implémentation de l'algorithme « de Rabin-Karp » lui-même. Ce second fichier `.c` devra utiliser une directive `#include` pour « inclure » un fichier d'entête associé au premier qui *expose* les *symboles* (en l'occurrence les noms de fonction) qui doivent pouvoir être utilisés. Ce fichier d'entête `hashp.h` devra être le suivant :

```
#ifndef HASHP_H // si le symbole préprocesseur HASHP_H
                // n'est pas défini
#define HASHP_H // définit le symbole préprocesseur HASHP_H
#include <stdint.h>
#include <stdlib.h>

uint32_t get_random_key(void);
uint32_t expp(uint32_t k, size_t n);
uint32_t hashp(uint32_t k, size_t n, uint8_t m[n]);
uint32_t hashp_inc(uint32_t k, uint32_t knmo, uint32_t h,
                   uint8_t old, uint8_t new);

#endif // fin du si
```

Les deux premières et la dernière ligne sont des *directives* pour le *préprocesseur* qui servent à éviter que les symboles ne soient recopiés plusieurs fois dans le fichier (ce qui en C déclencherait une erreur de compilation).

Conventionnellement, le fichier `hashp.c` devra définir les quatre fonctions appa-

raissant ici dans le fichier `hashp.h`, mais elle pourra également en définir d'autres qui auront vocation à lui rester *locales*, c'est à dire qui ne pourront pas être utilisées dans un autre fichier `.c` (même s'il est possible de tricher si l'on ne prend pas garde).
Pour résumer, notre programme sera constitué :

- du fichier `hashp.h` ci-dessus ;
- d'un fichier `hashp.c` qui devra implémenter les fonctions y apparaissant ;
- d'un fichier `fact_rk.c` qui « inclura » le fichier `hashp.h` (et pourra utiliser les fonctions qui y sont déclarées) ; le fichier `hashp.h` n'étant pas connu du compilateur, l'inclusion devra se faire comme :

  ```
  #include "hashp.h"
  ```

  si `hashp.h` se trouve dans le même répertoire que `fact_rk.c`, ou sinon en indiquant un (autre) chemin relatif ou absolu vers `hashp.h`.

Dans notre cas, pour simplifier, c'est aussi `fact_rk.c` qui définira la fonction `main`, mais celle-ci pourrait en principe se trouver dans son propre fichier `.c`.
La compilation du programme complet pourra dans un premier temps être faite simplement en donnant les différents fichiers `.c` (et uniquement ceux-là) en argument :

```
> cc [options de compilation] hashp.c fact_rk.c
```

cependant, l'un des avantages de séparer le programme en plusieurs fichiers est que ceux-ci peuvent être compilés séparément :

- chaque fichier `.c` (ici `file.c`) est compilé en un fichier `.o` avec la commande :

  ```
  > cc -c [options de compilation] file.c
  ```

- les fichiers `.o` (par exemple ici `file.o` et `file2.o`) sont tous *liés* entre eux avec la commande :

  ```
  > cc file.o file2.o
  ```

Ceci est intéressant car si un unique fichier `file.c` a été modifié, une nouvelle version du programme tout entier peut être compilée en recompilant uniquement `file.c` en `file.o` et en liant ce dernier aux autres fichiers `.o` (qui n'ont pas changé). Le temps de compilation pour de gros programmes peut alors être nettement réduit. Mais cela est-il une si bonne chose que ça ?
En pratique, on utilise des outils automatiques (comme `make`) afin de déterminer les étapes de recompilation à effectuer en fonction des modifications faites aux fichiers et de leurs dépendances (qui doivent être indiquées à `make`). Dans cet exercice on conseille cependant de compiler « à la main ».
Pour résumer, une compilation complète de notre programme pourra par exemple se faire comme :

```
> cc -c -Wall -Wextra hashp.c
> cc -c -Wall -Wextra fact_rk.c
> cc -o fact_rk hashp.o fact_rk.o
```

Fichier `hashp.c`

1. Définissez une variable globale `p` en lecture seule de type `uint32_t` et de valeur $2^{31}-1 = 2\,147\,483\,647$.
2. Écrivez une fonction C de signature :
   `uint32_t get_random_key(void)`
   qui renvoie un entier $\in [\![1, p-1]\!]$ tiré uniformément au hasard. (Faites au mieux étant donnés les capacité de génération d'aléa de votre système.)
3. Écrivez une fonction C de signature :
   `uint32_t mulp(uint32_t x, uint32_t y)`
   qui pour `x` et `y` $\in [\![0, p-1]\!]$ renvoie l'unique entier $\in [\![0, p-1]\!]$ qui est congru à leur produit modulo p.
   Attention aux dépassement de capacité.
4. **Cette question est facultative et n'est à traiter que si vous comptez finir l'exercice.**
   La fonction précédente est généralement compilée efficacement par les compilateurs modernes ; c'est à dire qu'elle exploite le fait que `p` est une « constante » connue à la compilation pour ne pas utiliser d'instruction de division et à la place utiliser des multiplications par des constantes bien choisies. Dans le cas précis de $p = 2^{31}-1$, il est cependant possible de faire mieux que la méthode utilisée les compilateurs usuels en implémentant une réduction « de Barrett » qui exploite le fait que :
   - $2^{31} \equiv 1 [p]$
   - le quotient dans la division par 2^{31} peut être calculée par un décalage de 31 bits vers la droite (avec `>> 31`)
   - le reste dans la division par 2^{31} peut être calculé par un ET bit-à-bit (avec `& 0x7FFFFFFF`)

   Alors, soit z un nombre dans un intervalle approprié que l'on cherche à « réduire modulo p », on peut efficacement calculer un $z' \in [\![0, 2p]\!]$ qui lui est congru comme $z' = z \div 2^{31} + z \mod 2^{31}$

   Écrivez une fonction C de signature :
   `uint32_t mulp_brt(uint32_t x, uint32_t y)`
   de mêmes spécifications que `mulp` et qui utilise l'algorithme esquissé. Testez-la et comparez ses performances avec `mulp` (en compilant *a minima* avec `-O2`).
5. Écrivez une fonction C de signature :
   `uint32_t expp(uint32_t k, size_t n)`
   qui pour $k \in [\![0, p-1]\!]$ calcule et renvoie l'unique valeur congrue à k^n modulo p (en définissant 0^0 comme 1).
6. Écrivez une fonction C de signature :
   `uint32_t hashp(uint32_t k, size_t n, uint8_t m[n])`

qui interprète `m` comme le polynôme $\sum_{i=0}^{n-1} \texttt{m}[i]X^{n-1-i}$ de $\mathbb{Z}/p\mathbb{Z}[X]$ et l'évalue en `k` en utilisant la méthode « de Horner ».

7. Testez (idéalement en écrivant un programme de test dans un autre fichier). Vous pouvez par exemple notamment utiliser le fait que les polynômes $234X^3+53X^2+12X+3$ & $129X^3+176X^2+222$ s'évaluent respectivement en 626 108 117 & 224 592 768 sur 647 451 855 et 540 957 163 & 857 813 228 sur 1 179 878 780.
8. Écrivez une fonction C de signature :
   ```
   uint32_t hashp_inc(uint32_t k, uint32_t knmo, uint32_t h,
                      uint8_t old, uint8_t new)
   ```
   qui pour `h` égal à l'évaluation en `k` d'un polynôme $\sum_{i=0}^{n-1} r_i X^i$ avec r_{n-1} égal à `old` et `knmo` égal à $\texttt{k}^{n-1}$ « modulo p » calcule efficacement et renvoie l'évaluation en `k` du polynôme $\texttt{new}+\sum_{i=0}^{n-2} r_i X^{i+1}$.
9. Testez. Vous pouvez par exemple notamment utiliser le fait que les polynômes $234X^3+53X^2+12X+3$, $53X^3+12X^2+3X$ et $12X^3+3X^2+114$ s'évaluent en 52 023 785 respectivement à 1 023 609 573, 1 354 551 223 et 2 052 118 721.

Fichier `fact_rk.c`

10. Écrivez une fonction C de signature :
    ```
    size_t fact_rk(size_t fl, char f[fl],
                   size_t sl, char s[sl])
    ```
    qui utilise l'algorithme « de Rabin-Karp » instancié avec la fonction de hachage implémentée dans `hashp.c` pour renvoyer `-1` si la chaîne de caractères `f` n'est pas un facteur de `s`, et sinon l'indice de sa première occurrence.
    (Vous devriez pouvoir ignorer les avertissement de compilation relatifs à l'interprétation de valeurs de type `char *` comme valeurs de type `uint8_t`.)
11. Testez.