TP n°25 - Algorithme de Rabin-Karp

Toutes les fonctions de ce TP seront écrites dans un fichier nommé `NOM_rabin-karp.c`.
Dans tout le TP, on utilisera la fonction de hachage de conversion de base présentée lors du précédent TP :

$$\begin{array}{rccl} h: & \mathcal{S} & \longrightarrow & [\![0,\, m-1]\!] \\ & c_0 \dots c_{p-1} & \mapsto & \displaystyle\sum_{j=0}^{p-1} \text{code\_ascii}(c_j) \times b^{p-1-j} \bmod m \end{array}$$

où m et b seront explicités dans ce contexte particulier. On peut commencer avec $m = 1023$ et $b = 31$.

Attention, ici, par rapport au TP précédent, on calcule bien les puissances à l'envers... ce qui va bien avec Horner finalement !

Pour gérer correctement les caractères accentués, on fera un *cast* `unsigned char` lors de la récupération du code ASCII d'un caractère car les caractères français accentués font partie de la table ASCII étendu qui va au-delà de la valeur 128, donc avec un bit de poids fort à 1 sur 8 bits, ce qui serait interprété comme une valeur négative sans ce cast.

L'algorithme de Rabin-Karp est un algorithme de recherche d'un motif (*pattern*) dans un texte. Dans toute la suite, on notera n le nombre de caractères total du texte, et p le nombre de caractères du motif. **Attention, p est la taille du motif, m l'entier utilisé pour les congruences.**

Au lieu de faire des comparaisons lettre à lettre comme dans la recherche naïve, l'algorithme de Rabin-Karp va utiliser une fonction de hachage pour calculer des empreintes sur des séquences de caractères du texte de même taille que celle du motif, et les comparer avec l'empreinte calculée sur le motif.

L'algorithme calcule l'empreinte des p premiers caractères du texte $c_0 \dots c_{p-1}$ et la compare à celle du motif recherché, qui a été pré-calculée au début de l'algorithme.

Si l'empreinte correspond à celle du motif, les propriétés de pseudo-injectivité de la fonction de hachage permettent d'espérer une occurrence du motif : on le vérifie alors avec une comparaison caractère par caractère.

Par contre, si les deux clés sont différentes, on est absolument certains qu'il ne peut y avoir de concordance entre les p caractères du texte et le motif. On décale alors l'analyse d'un caractère et on recommence, en calculant l'empreinte de $c_1 \dots c_p$...etc et ceci jusqu'à atteindre la fin du texte.

Vous le comprenez bien, dès que la taille du motif p sera grande, les puissances b^j vont nous amener à manipuler de très grands entiers. **On va donc de suite s'autoriser à travailler uniquement avec des entiers 64 bits signés `int64_t` dans nos calculs.**

Vous n'utiliserez aucun entier non signé dans votre code, pour éviter toute difficulté subtile.

Exercice 1 (Algorithme de Rabin-Karp naïf).

1. Rappeler rapidement l'algorithme de recherche naïve d'un motif dans un texte et rappeler la complexité de cet algorithme.
2. Coder la fonction de hachage `int64_t hash(char *k, int64_t p, int64_t m)` qui prend également en entrée la taille p de la chaîne de caractères à hacher. N'oubliez pas le cast sur les codes ASCII.
3. Coder l'algorithme de Rabin-Karp sous la forme d'une fonction `rabin_karp_naive`.
   On pourra utiliser la fonction `strncmp(char *s1, char *s2, int p)` de la bibliothèque `string`, qui mesure la différence lexicographique entre les deux chaînes `s1 s2` sur leurs p premiers caractères(allez lire le manuel, commande `man`).
   On pourra également utiliser une astuce comme celle-ci :

   ```
   char *s = &(text[i])
   ```

   pour considérer la chaîne de caractères qui démarre au niveau du caractère d'indice `i` de la chaîne `text` et éviter d'avoir à dupliquer des caractères pour pouvoir faire des comparaisons.
4. Testez votre algorithme sur de petits textes et motifs donnés en ligne de commande.
5. Quelle est la complexité de votre fonction dans le meilleur des cas ? Dans le pire des cas ? *Indication : n'oubliez pas le coût de calcul de l'empreinte !*

En l'état, cet algorithme de Rabin-Karp n'est pas très avantageux par rapport à la recherche naïve, même dans le meilleur des cas, où très peu d'empreintes correspondent à celle recherchée.

Exercice 2 (Exponentiation rapide modulaire).
Nous allons avoir besoin d'un algorithme d'exponentiation modulaire, c'est-à-dire un algorithme qui calcule $x^n \bmod m$ pour $x \in \mathbb{N}$, $n \in \mathbb{N}$ et $m \in \mathbb{N}$ donnés.

1. Implémenter l'algorithme d'exponentiation rapide `pow_bin` [a] classique sous forme itérative (déjà fait, question de cours).
2. Copier la fonction précédente pour créer une nouvelle fonction `int64_t power_bin_mod(int64_t x, int64_t j, int64_t m` et adaptez-la pour qu'elle effectue cette fois-ci une exponentiation modulaire sur de grands entiers x encodés sur 64 bits.
3. Tester votre fonction sur de petites valeurs et en les vérifiant à la main.

a. *bin* pour *binary*, car l'algorithme est dichotomique...

Exercice 3 (Algorithme de Rabin-Karp amélioré - Fonction de hachage déroulante).
Pour améliorer la complexité temporelle de l'algorithme de Rabin-Karp, il suffit de se rendre compte que, pour la fonction de hachage polynomiale choisie, le calcul de l'empreinte $h(c_{i+1} \dots c_{i+p})$ peut être effectué à partir de l'empreinte $h(c_i \dots c_{i+p-1})$ précédente. On dit que la fonction de hachage est **déroulante**.

1. Expliciter cette formule mathématique liant le calcul de l'empreinte $h(c_{i+1} \dots c_{i+p})$ à celle de l'empreinte $h(c_i \dots c_{i+p-1})$.
2. On suppose que l'on n'effectue pas de `mod m`, sur le produit $\text{code\_ascii}(c_i) \times b^{p-1}$. Montrer théoriquement que, pour n'importe quel choix de m, même très grand, et même avec un stockage 64 bits la soustraction peut donner un résultat négatif. On donnera une condition suffisante sur m et p pour caractériser les situations problématiques.
3. Créer une nouvelle version `rabin_karp` de votre algorithme en plaçant correctement les modulos pour éviter la création d'empreintes négatives (attention, les *cast* sauvages sont interdits !) et testez la sur des motifs de quelques lettres. *Attention : Le terme $b^{p-1} \bmod m$, coûteux car il fait intervenir une congruence sur de grandes valeurs, est indépendant de j et n'a pas besoin d'être recalculé à chaque tour de boucle. Il sera donc pré-calculé de manière efficace, une bonne fois pour toute, en début d'algorithme.*
4. Donner la complexité de votre nouvel algorithme dans le meilleur et dans le pire des cas.
5. Rajouter des vérifications dans votre algorithme pour vérifier que les empreintes calculées par la formule déroulante sont positives et identiques à celles calculées directement par votre ancienne fonction de hachage.
6. Tester à nouveau votre algorithme en fonction grossir la taille du motif p. Qu'observez vous ?

On rappelle la formule déroulée trouvée à l'exercice précédent, permettant de calculer la nouvelle empreinte décalée d'un caractère à partir de l'empreinte précédente :

$$h(c_{i+1} \dots c_{i+p}) = \big(b \times (h(c_i \dots c_{i+p-1}) - \text{code\_ascii}(c_i) \times b^{p-1}) + \text{code\_ascii}(c_{i+p})\big) \bmod m$$

Exercice 4 (Résolution du problème des valeurs négatives liées au débordement mémoire).
L'opérateur modulo `x % m` est implémenté en C de sorte à préserver le signe de x, ce qui n'est pas la définition mathématique du reste de la division euclidienne.
Cela peut poser problème dans toutes vos fonctions, par exemple dans le calcul de la fonction de hachage. Dans l'algorithme de Rabin-Karp dans le cas, par exemple, où le produit $b^{p-1} \times \text{code\_ascii}(c_0)$ crée un débordement mémoire. Dans ce cas, ce terme devient négatif et l'évaluation machine de

$$h(c_i \dots c_{i+p-1}) - \left(b^{p-1} \times \text{code\_ascii}(c_0)\right)$$

peut elle même déborder et devenir négative, ce qui amener une valeur d'empreinte négative incohérente, alors même que le terme de droite est toujours censé être inférieur au terme de gauche.
Pour pallier ce problème, écrire un opérateur `int64_t modulo(int64_t x, int64_t m)` utilisant l'opérateur `%` qui calcule effectivement un reste **positif** modulo une valeur, conforme à la définition mathématique.
Dans la suite, tous les appels à l'opérateur `%` devront être remplacés par cette fonction améliorée pour valider la justesse mathématique de nos calculs.

Exercice 5 (Optimisation de la fonction de hachage).
On peut améliorer les performances de l'algorithme sur de très gros textes en travaillant sur la fonction de hachage.

1. Quel premier choix sur la fonction de hachage, très simple à réaliser peut-on faire pour tenter de renforcer injectivité de la fonction de hachage ?
2. Dans l'algorithme de Rabin Karp, l'usage de la fonction de hachage diffère à plusieurs égards de celle du TP précédent. Expliquez clairement ces différences de contexte, et dites en quoi cela augmente notre marge de manœuvre pour le choix de m.
3. On peut donc « lâcher » sur m puisqu'il n'y a aucun enjeu d'allocation mémoire dans ce contexte, et que l'on a même tout intérêt à étendre au maximum l'intervalle des empreintes $[\![0, m-1]\!]$ pour renforcer l'injectivité et limiter les collisions, c'est-à-dire les « faux positifs » dans l'algorithme de Rabin-Karp. On va donc aller dans le sens de l'injectivité en poussant la valeur de m le plus loin possible, dans l'idéal jusqu'au maximum de la représentation machine des entiers signés positifs 64 bits, soit $2^{63} - 1$

   Pour ne pas tricher et continuer notre analyse rigoureuse sur les signes, on reste pour cette question avec des entiers signés.

   a. Nous l'avons vu, les débordements mémoire arrivent vite et entraînent la création de valeurs négatives par overflow (le bit de poids fort passe à 1). Nous avons corrigé ce problème en créant un nouvel opérateur modulo. Toutefois, celui-ci crée un surcoût et nous avons tout intérêt à éviter ces situations. En analysant toutes les opérations effectuées dans l'intégralité de votre code, proposez une choix de m optimal et qui limite les débordements mémoire et la création de valeurs négatives.

   b. Valider le bon fonctionnement de votre nouveau code, y compris sur de très grands motifs.

Exercice 6 (Recherche de motif réaliste).
L'intégralité du texte *Le tour du monde en 80 jours*, de Jules Verne, est disponible sur Moodle au format texte ASCII.
Coder une fonction `int grep(char *path, char *pattern)` permettant d'obtenir le nombre d'occurrences d'un mot de votre choix dans ce texte et une valeur négative s'il y a eu un problème.