Algorithmes et Complexité
Projet

1 Recherche de chaînes

Le but de cet exercice est de créer un logiciel qui cherche un gros fichier f à l'intérieur d'un gros fichier F.

On rappelle le principe de fonctionnement. Soit n la taille de f et N la taille de F (en octets)

Soit p un nombre premier tiré aléatoirement.

Soit c_0, c_1, c_{n-1} les octets du fichier f et $d_0, d_1 \dots d_N$ les octets du fichier F_1.

Le *fingerprint* du fichier f est obtenu de la façon suivante :

$$finger_p(f) = (c_0 \times 256^{n-1} + c_1 \times 256^{n-2} + \cdots + c_{n-2} \times 256 + c_{n-1}) \mod p$$

Pour tester si f apparaît dans F, on calcule tous les fingerprints obtenus en regardant n caractères consécutifs du fichier F.

Autrement dit, on calcule toutes les fingerprints suivants :

$$finger_p(F, k) = (d_k \times 256^{n-1} + d_{k+1} \times 256^{n-2} + \cdots + d_{n-2} \times 256 + d_{k+n-1}) \mod p$$

pour toutes les valeurs de k entre 0 et $N - n$.

Pour savoir si le fichier f apparaît dans F, il suffit alors de tester si une des fingerprints $finger_p(F, k)$ est égale à la fingerprint $finger_p(f)$.

Le sujet est en deux parties. Pour avoir une note correcte, il est attendu d'avoir traité a minima toutes les questions de la première partie.

1.1 Première partie - entiers 32 bits

Dans cette partie, on choisira un nombre p premier entre 3 et $2^{23} - 1 = 8388607$. En particulier on pourra utiliser le type *int* de Java.

Une des difficultés des opérations est que si x et y sont entre 0 et $2^{23} - 1$ alors $(x \times y)$ mod p ne donne pas le bon résultat : En effet le produit $x \times y$ nécessite plus de 32 bits, donc sera tronqué lors du calcul.

On pourra utiliser le fragment de code suivant pour calculer le produit de deux nombres :

```
int multiply(int x, int y, int p)
{
   int result = 0;
   while (y)
     {
        if (y & 1) result = (result + x) % p;
        y >>= 1;
        x = (2*x)%p;
     }
   return result;
}
```

Ce fragment de code étant plus lent qu'une multiplication classique, on évitera de l'utiliser autant que possible.

1.1.1 Tirer un nombre premier au hasard

Pour tirer un nombre premier au hasard, on tire un nombre au hasard, et on vérifie s'il est premier. Si ce n'est pas le cas, on recommence. Pour l'exercice, il n'est pas nécessaire d'être certain qu'un nombre est premier : un nombre pseudo-premier est amplement suffisant. On testera donc uniquement si $2^{p-1} \mod p = 1$: Sur les 8388604 nombres qu'on va tester, le résultat est faux uniquement sur 1000 d'entre eux (dont l'entier 2).

Q 1) Ecrire une fonction `int puissance(int x,k,p)` qui calcule $x^k \mod p$. Par exemple, `puissance(9,118,17)` doit renvoyer 4. Le calcul de $x^k \mod p$ doit mettre de l'ordre de $O(\log k)$ étapes. On pourra utiliser la fonction `multiply`

Q 2) Ecrire une fonction `bool pseudoprime(int p)` qui teste si p est pseudo-premier.

Q 3) Ecrire une fonction `int nextprime()` qui renvoie un nombre premier tiré aléatoirement entre 2 et $2^{23} - 1$

1.1.2 Calcul du fingerprint $finger_p(f)$

Q 4) Ecrire une fonction `fingerprint(int p,String fn)` qui calcule le fingerprint du fichier `fn`. Pour le fichier `test1` fourni, le fingerprint pour $p = 5407$ doit être de 2123. Il n'est pas nécessaire d'utiliser la fonction `multiply` ou `puissance` pour cette question.

1.1.3 Calcul des fingerprint $finger_p(F, k)$

Q 5) Ecrire une fonction `find(int p, String fn, int fp, int n)` qui calcule tous les fingerprints $finger_p(F, k)$ pour toutes les valeurs de k entre 0 et $N - n$ et renvoie vrai si l'une d'entre elles vaut fp. Attention : Une fois qu'on a calculé $finger_p(F, 0)$, il ne faut pas refaire tous les calculs pour trouver $finger_p(F, 1)$, $finger_p(F, 2)$, etc. Un appel à la fonction `puissance` est autorisé

Q 6) Assembler toutes les fonctions en un programme qui répond à la tâche demandée.

Q 7) Tester le programme sur plusieurs fichiers.

2 Deuxième partie - Compléments

Q 8) Expliquer le code de la fonction `multiply`.

Q 9) Pourquoi a-t-on choisi des nombres premiers entre 2 et et 2^{23}et non pas entre 2 et 2^{31} ?

Q 10) Lancer une centaine d'exécutions de votre programme avec comme entrée `test3.xpm` et `test4`. Que constatez-vous ?

Q 11) Le fichier `test2` a été conçu de sorte que son fingerprint soit égal à 0 pour tout nombre premier p inférieur à 2^{23}. Expliquer comment un tel fichier a été fabriqué.

Q 12) Utiliser le fichier `test2` pour transformer le fichier `test5.xpm` en un fichier `fool.xpm` qui est différent du fichier de départ mais qui a les mêmes fingerprint.

Q 13) Passer votre code en 64 bits en transformant tous les int en long. On choisira désormais un nombre premier entre 2 et 2^{55}.

Q 14) Vérifier que votre nouveau programme trouve des fingerprint différentes de 0 pour les fichiers `fool.xpm` et `test5.xpm`.