Algorithmes et Complexité
Projet

Le devoir doit être effectué par groupe de 1 *à* 2 *personnes, chaque personne appartenant à exactement un groupe.*

1 Travail à effectuer

Description

Le devoir est à rendre sur `arche` pour le **8 janvier 2018 à 8h**.

Le travail à effectuer est en deux parties complètement indépendantes.

Pour chacune des parties, vous devez rendre :

— Les sources de vos programmes
— Les graphiques obtenus
— Un rapport répondant aux différentes questions, et qui explique en particulier la répartition du travail dans le groupe

Vous pouvez bien entendu incorporer les graphiques dans le rapport.

Le projet étant un projet d'algorithmique et non pas de développement logiciel, la majeure partie de la notation portera sur l'implémentation correcte des algorithmes et les réponses aux questions.

Graphiques

La deuxième partie nécessite de produire un graphique permettant d'illustrer les fonctions.

Il est conseillé, mais ce n'est pas nécessaire, d'utiliser l'outil gnuplot pour produire les graphiques.

Si le fichier `toto.dat` contient deux nombres par ligne, le programme `toto.plot` suivant (qu'on lance par exemple avec `gnuplot toto.plot`) vous produira un fichier pdf que vous pourrez incorporer à votre rapport :

```
set terminal pdf
set output "toto.pdf"
set xlabel "X axis"
set ylabel "Y axis"
set title "Un nuage de points"
set grid

plot "toto.dat" with points title "premier"
```

La lecture du manuel d'utilisation est conseillée. Un graphique sans légende, et sans unité, sera très fortement pénalisé.

2 Recherche de chaînes

Le but de cet exercice est d'implémenter un concept de fingerprint, type md5 ou sha, mais de façon plus simple en utilisant de l'arithmétique.

De cette façon, on pourra savoir si deux fichiers sont égaux en comparant leurs fingerprints.

Le principe du fonctionnement est le suivant.

On se donne un nombre premier p de 24 bits tiré aléatoirement.

Soit f un fichier de n octets. on va voir f comme un grand nombre écrit en base 256. Plus exactement, si le fichier est constitué des octets c_0, c_1, c_{n-1}

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

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

Par exemple, considérons le fichier constitué du message `bonjour`. Les caractères `b,o,n,j,o,u,r` sont respectivement de code ASCII 98, 111, 110, 106, 111, 117, 114, donc si $p = 17$, il faut calculer

$$(98 \times 256^6 + 111 \times 256^5 + 110 \times 256^4 + 106 \times 256^3 + 111 \times 256^2 + 117 \times 256 + 114) \mod 17 = 2$$

Bien entendu, il ne faut pas calculer la somme puis faire l'opération modulo, puisque la somme est beaucoup trop grande, il faut s'y prendre autrement.

Dans toute la suite, le nombre p premier qu'on choisira sera 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, il ne faut l'utiliser que lorsqu'il est nécessaire.

2.1 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 $2^{23} - 1$

2.2 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.

Q 5) Tester le programme sur plusieurs fichiers : Prenez deux fichiers différents et calculer leur fingerprint pour le même nombre premier. Vérifier expérimentalement que, avec grande probabilité, les fingerprints obtenus sont différents.

Q 6) Lancer une centaine d'exécutions de votre programme avec comme entrée `test3.xpm` et `test4`. Que constatez-vous ?

Q 7) 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 8) Essayez d'expliquer comment le fichier `test4` a été obtenu.

Q 9) (Bonus) 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.

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

Le fingerprint peut être utilisé pour savoir si un fichier f (de taille n) est contenu dans un fichier F plus grand : On se donne un entier p choisi aléatoirement, on calcule le fingerprint de f et on calcule le fingerprint de tous les blocs de taille n à l'intérieur de F.

Pour que cela soit efficace, il ne faut pas tout recommencer : il faut trouver comment calculer le fingerprint du bloc suivant (celui allant des positions $p + 1$ à $p + n$) connaissant le fingerprint du bloc courant (celui allant de p à $p + n - 1$).

Q 10) Ecrire un programme qui répond à la tâche demandée.

Q 11) Tester le programme sur plusieurs fichiers.

3 Problème de bin packing

On dipose de n objets, $i = 1, \ldots, n$ à ranger dans des boites. Pour chaque objet i, on a une hauteur h_i (un entier positif), et chaque boîte a une hauteur H. Le but est de ranger les objets dans un nombre minimum de boites.

Q 12) Implémenter une fonction `FractionalPacking` qui étant donné une liste d'objet et une taille de boîte retourne le nombre de boîtes utilisées en supposant qu'il est possible de découper les objets.

Dans la suite, on s'intéressera à des objets insécables. Vous pouvez représenter un remplissage par une liste de tuples (taille occupée, liste des tailles des objets dans la boîte), par exemple (10, [6 ; 3 ; 1]).

Q 13) Implémenter une fonction `FirstFitPacking` qui étant donné une taille des boîtes et une liste d'objets, retourne un remplissage où chaque élément a été ajouté à la première boîte capable de l'accueillir.

Q 14) Implémenter une fonction `BestFitPacking` qui étant donné une taille des boîtes et une liste d'objets, retourne un remplissage où chaque élément a été ajouté à la boîte dont la capacité restante est la plus grande après insertion de cet éélement.

Q 15) Implémenter deux nouvelles fonctions `FirstFitDecreasingPacking` et `BestFitDecreasingPacking`, utilisant les fonctions `FirstFitPacking` et `BestFitPacking` où les objets sont triés dans l'ordre décroissant de leur taille.

Q 16) Tester vos 4 méthodes et la méthode `FractionalPacking` en fonction du nombre d'objets ($n = 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000$). Pour cela, pour chaque taille n, on générera 20 instances aléatoires avec des boîtes de hauteur exactement $1.5n$ et des objets de tailles choisies aléatoirement dans l'intervalle $[0.2n, 0.8n]$ et on calculera, pour chaque taille n et chaque méthode, la moyenne sur les 20 instances du nombre de boîtes utilisées. Les données devront être représentées sur un graphique, et le graphique devra être commenté.