Université de Limoges

# Développement Logiciel Cryptographique

## TP n° 3

## 1 Test de Fermat

1. Écrivez une fonction `test_fermat_base` qui prend en entrée deux entiers $n$ et $a$ et qui teste le critère de Fermat sur l'entier $n$ pour la base $a$.
2. Écrivez une fonction `test_fermat` qui prend en entrée deux entiers $n$ et $t$ et qui décide si l'entier $n$ est composé ou probablement premier sur la base du test de Fermat tenté avec $t$ bases dont les valeurs sont choisies aléatoirement entre 2 et $n-2$.
3. Écrivez un programme qui prend en entrée une taille $b$, un nombre de bases $t$ et un nombre d'entiers testés $r$ spécifiés par l'utilisateur. Ce programme génèrera $r$ entiers de taille exactement $b$ bits et appellera pour chacun d'eux la fonction `test_fermat` pour tester leur primalité avec $t$ bases. Utilisez ce programme pour vérifier expérimentalement le théorème sur la densité des nombres premiers [1]. Par exemple, obtenez une estimation de la densité de premiers parmi les entiers de taille 512 ou 1024 bits.
4. Écrivez un programme qui prend en entrée une taille $b$, un nombre de bases $t$ et un nombre de premiers générés $r$ spécifiés par l'utilisateur. Ce programme générera des entiers de taille exactement $b$ bits et appellera pour chacun d'eux la fonction `test_fermat` pour tester leur primalité avec $t$ bases. Le programme devra s'arrêter lorsque $r$ nombres auront été déclarés premiers par votre test. Pour chacun des nombres déclarés

1. La densité des nombres premiers autour de $x$ tend asymptotiquement vers $\frac{1}{\ln x}$.
Autrement dit, la proportion de nombres premiers parmi les entiers de taille $b$ bits est approximativement $\frac{1}{\ln 2^b} = \frac{1}{b \ln 2}$.

premiers, vous vérifierez qu'il l'est effectivement [2] et l'afficherez à l'écran si tel n'est pas le cas.

# 2 Test de Miller-Rabin

1. Écrivez deux fonctions `test_miller_rabin_base` et `test_miller_rabin` qui sont les adaptations des fonctions `test_fermat_base` et `test_fermat` pour le critère de Miller-Rabin.

2. Adaptez le programme de l'exercice 4 de la section 1 qui affichait les pseudo-premiers de Fermat pour qu'il affiche maintenant les pseudo-premiers forts. Comparez les nombres de pseudo-premiers de chacun des deux types à paramètres $b$, $t$ et $r$ identiques.

# 3 Génération de premiers par la méthode du crible simple

1. Écrivez une fonction qui prend en entrée trois entiers $b$, $k$ et $t$, et qui génère un entier premier de taille exactement $b$ bits par la méthode du crible simple qui permet de s'assurer qu'un candidat n'est divisible par aucun des $k$ plus petits premiers avant d'être testé par la méthode de Miller-Rabin avec $t$ bases.

   Astuce : Afin de ne pas calculer tous les petits premiers à chaque appel de votre fonction, il pourrait être utile de les faire calculer une seule fois par l'appelant qui les transmettrait en paramètre à votre fonction.

2. Écrivez un programme qui prend en entrée une taille $b$, un nombre de bases $t$, un nombre de petits premiers $k$ et un nombre de premiers générés $r$ spécifiés par l'utilisateur. Ce programme générera $r$ nombres premiers de $b$ bits exactement par la méthode du crible simple sur les $k$ plus petits premiers.

   Pour un même nombre de premiers générés, observez comment varie le temps d'exécution en fonction de $k$. Par exemple, utilisez les valeurs $b = 128$, $t = 3$ et $r = 1000$.

   Astuce : En lançant votre programme, en faisant simplement précéder le nom de votre programme par la commande `time`, vous obtenez en

2. À cette fin vous pourrez utiliser le test de primalité de GMP et l'assimiler à un test de primalité prouvée lorsque le nombre de bases est égal à 10.

fin d'exécution des informations sur le temps qu'il a mis à s'exécuter.
Exemple : `$ time crible_simple 128 3 10 1000`

Astuce encore meilleure : Vous pouvez obtenir le même type d'information directement dans votre programme. Pour cela il vous faut inclure les fichiers systèmes `sys/types.h` et `sys/resource.h`, et intégrer dans votre code la fonction suivante qui retourne à chaque appel le temps "user", exprimé en milli-secondes, qui s'est écoulé depuis le début de l'exécution.

```
unsigned long int cputime()
{
  struct rusage rus;

  getrusage (0, &rus);
  return rus.ru_utime.tv_sec * 1000 + rus.ru_utime.tv_usec / 1000;
}
```

3. Vous êtes maintenant en mesure de modifier votre programme afin qu'il répète en boucle la génération des $r$ nombres premiers pour des valeurs croissantes de $k$ pour chacune desquelles le temps d'exécution de la boucle est mesuré. Stockez dans un fichier des lignes comprenant simplement la valeur de k suivie du temps d'exécution. Utiliser le programme `gnuplot` pour visualiser de manière graphique l'évolution du temps d'exécution. Pour cela, lancez `gnuplot` dans une fenêtre terminal et fournissez des commandes à son interpréteur :
Exemple : `gnuplot> plot "timing.txt" with lines`

   Astuce (encore une !) : Dans votre programme, plutôt que d'ouvrir explicitement un fichier, puis écrire dedans, puis enfin le refermer, vous pouvez afficher vos informations à l'écran (sur `stdout` ou `stderr`) puis rediriger cette sortie vers un fichier lors de l'exécution.

# 4 Génération de premiers par la méthode du crible optimisé

1. Ajoutez dans votre programme une fonction qui génère les premiers par la méthode du crible optimisé. Dans cette variante le test de divisibilité de chaque candidat par l'ensemble des petits premiers se fait sans division explicite.

   Expérimentez et comparez les temps d'exécution obtenus avec les deux variantes pour différents jeux de paramètres.