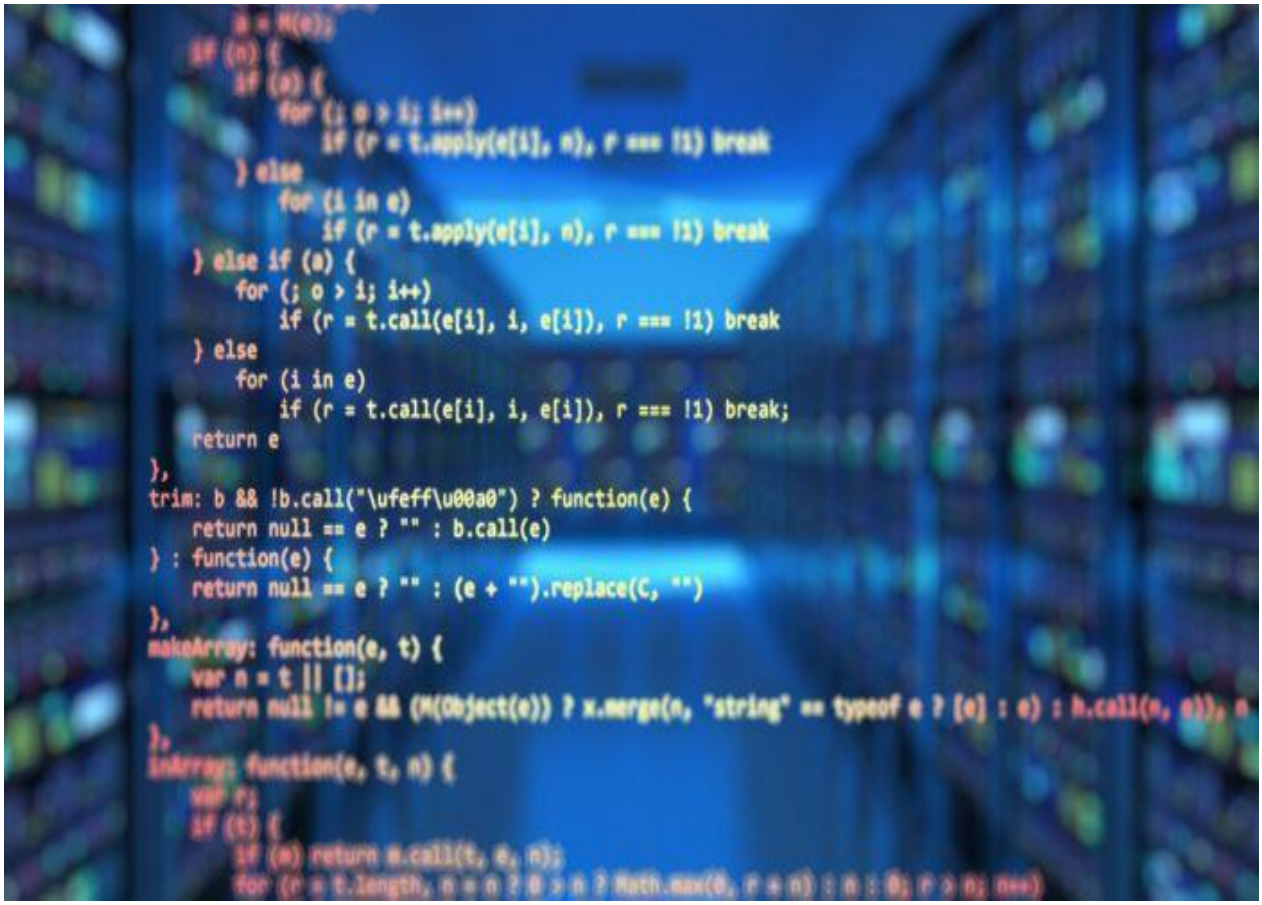
Maths Discrètes

TP Noté

Année 2020 – 2021



Réalisé par :

FERCHIOU Iskander
OULED MOUSSA Yanis

Responsable de l'UE :

M. GAVIN Gérald

POLYTECH Lyon - UNIVERSITE CLAUDE BERNARD LYON 1
Domaine Scientifique de La Doua – 15, Boulevard Latarjet
69622 VILLEURBANNE CEDEX
Tél. (33) 04.72.43.12.24 - Fax. (33) 04.72.43.12.25
http://polytech.univ-lyon1.fr/

Exercice 1 : Euclide étendu

Algorithme d'Euclide étendu :

```
EuclideEtendu(7,6) = [1,1,-1]
```

Exercice 2 : Exponentiation modulaire

Algorithme de l'exponentiation modulaire :

```
ExpMod(4 , 13, 497) = 445
```

Exercice 3 : Primalité naïf

Algorithme de primalité naïf :

```
PrimaliteNaif(15) = false
```

Exercice 4 et 5 : Proportion de nombre premiers (PrimaliteNaif) et taux d'erreurs (TestFermat)

Exo 4 :

Plus la taille de l'intervalle $N \cap [2^i ; 2^{i+1}]$ est grande, plus la proportion de nombre premiers est petite.

Exo 5 :

Pour l'intervalle [2,4[, le taux d'erreur est de 50% car le chiffre 2 n'est pas considéré comme par notre test de Fermat. En effet, $2^1 \, mod \, 2 = 0$, ce qui est différent de 1. Concernant le reste des intervalles, le taux d'erreurs reste nul jusqu'à ce que i = 8 où l'on commence à avoir une valeur supérieure à 0.

Exécution du programme :

---------- Intervalle [2, 4[----------
Proportion naïve : 100.0%
Taux d'erreur de Fermat : 50.0%
---------- Intervalle [4, 8[----------
Proportion naïve : 50.0%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [8, 16[----------
Proportion naïve : 25.0%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [16, 32[----------
Proportion naïve : 31.25%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [32, 64[----------
Proportion naïve : 21.875%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [64, 128[----------

Proportion naïve : 20.3125%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [128, 256[----------
Proportion naïve : 17.96875%
Taux d'erreur de Fermat : 0.0%
---------- Intervalle [256, 512[----------
Proportion naïve : 16.796875%
Taux d'erreur de Fermat : 0.390625%
---------- Intervalle [512, 1024[----------
Proportion naïve : 14.6484375%
Taux d'erreur de Fermat : 0.390625%

Exercice 6 : Rapidité du testFermat par rapport à PrimaliteNaif

Protocole expérimental :

Pour chaque intervalle $N \cap [2^i ; 2^{i+1}]$, on compare le temps en millisecondes (ms) avant et après l'exécution de chaque code, que ce soit PrimilateNaif ou TestFermat. Etant donné que nous mesurons en ms, les premières valeurs entre Fermat et PrimaliteNaif ne sont pas significativement différentes.

Cependant, sur le long terme (donc sur des grands intervalles), on observe que le temps d'exécution de PrimaliteNaif est bien plus long que celui de Fermat.

Remarque : la valeur totale affichée est la somme de tous les temps d'exécution de chaque intervalle.

Exécution du programme :

Temps d'exécution en millisecondes (pour chaque intervalle $N \cap [2^i ; 2^{i+1}]$) :

---------- Naïf ----------
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 7.0, 5.0, 18.0, 102.0, 435.0, 1225.0]
Total : 1793ms
---------- Fermat ----------
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 4.0, 4.0, 5.0, 16.0, 16.0]
Total : 46ms

Exercice 7 : GenPremiers

```
GenPremiers(8) = 239
Est premier : true
```

Exercice 8 : PhiToFact

$phi(n) = phi(pq) = (p-1)(q-1)$. Or, l'algorithme polynomiale retourne les valeurs de p et q donc s'il n'existe pas d'algorithme polynomiale de factorisation, il n'existe pas d'algorithme polynomial A tel que A(n) -> phi(n).

Complexité : $O(n^2)$

```
PhiToFact(6) = [3, 2]
```

Exercice 9 : VerifPhi

Cette fonction vérifie que $phi(pq) = (p-1) * (q-1)$

```
VerifyPhi((int)fact[0], (int)fact[1]) = true
```

Exercice 10 : Vérification expérimentale de $(x^e)^d \equiv x \; mod \; n$

Preuve :

$$\begin{aligned}(x^e)^d &= x^{1 \; mod \; phi(n)} \\ &= x^{1 + k * phi(n)} \\ &= x * x^{k * phi(n)}\end{aligned}$$

D'après le théorème d'Euler, $x^{(k * phi(n))} \equiv 1 \; mod \; n$.
Par conséquent, $(x^e)^d \equiv x \; mod \; n$

Protocole expérimental :

Pour vérifier que $(x^e)^d$ est congru à x mod n :

- Etape 1 : on génère deux entiers aléatoires, p et q, de k bits.
- Etape 2 : on calcule $n = p * q$ et on récupère $phi(n)$.
- Etape 3 : on prend un e aléatoire qui n'est pas multiple de n et inversible modulo phi(n).
- Etape 4 : on vérifie que la formule est applicable sur un échantillon (par exemple un intervalle de taille 100).
- Etape 5 : Si un nombre x appartenant à l'intervalle est premier avec n, on test si $(x^e)^d$ est congru à x mod n. Cette relation doit toujours être vérifiée, ce qui est bien le cas.

Exécution du code :

$Factorisation : [113, 79]$
$n : 8927 - phi(n) : 8736 - inverse : 7327 - e : 31$
$Vérification \; de \; la \; formule : true$

Exercice 11 : $A_1(n_1, n_2, n_3, e, M_1, M_2, M_3)$

Les exercices de 11 à 14 traitent du chiffrement RSA.

La complexité de cette fonction est $O(2^n)$. Par conséquent, il ne s'exécute donc pas en temps polynomial.

Exécution du code :

$Ensemble\ p :\ [53, 59, 43, 37, 41, 61]$
$Ensemble\ n :\ [3127, 1591, 2501]$
$n :\ 1591$
$m\ témoin :\ 677$
$e :\ 257$
$Ensemble\ M :\ [2645, 27, 336]$
$m :\ 677$

Exercice 12 : $A_2(n_1, n_2, n_3, e, M_1, M_2, M_3)$

Dans ce cas, $p1 = p2 = p3$ donc $pgcd(n1, n2) = pgcd(n2, n3) = pgcd(n1, n3)$.
Il suffit donc de se servir de seulement 2 des n_i.

Le calcul du $pgcd(n_1,\ n_2) = p1\ et\ \frac{n_1}{p_1} = p2$.

Nous trouvons ensuite rapidement $phi(pq) = (p-1) * (q-1)$

Finalement, nous nous servons de la formule des M_i pour trouver m en nous servant de l'inverse modulaire de e par rapport à phi.

Cela permet de trouver m en temps polynomial.

Exécution du code :

$Ensemble\ p :\ [59, 61, 59, 41, 59, 43]$
$Ensemble\ n :\ [3599, 2419, 2537]$
$n :\ 2419$
$m\ témoin :\ 350$
$e :\ 1807$
$Ensemble\ M :\ [229, 52, 1468]$
$m :\ 350$

Exercice 13 : $A_3(n_1, n_2, n_3, e, M_1, M_2, M_3)$

Lorsque e est suffisamment petit, ici $e = 3$, il est possible de faire une attaque de Hastad en appliquant le Théorème des restes chinois pour obtenir un algorithme qui s'exécute en temps polynomial.

Preuve :

Dans ce cas on a : $M_i = m_i^3\ mod\ n_i$
En appliquant le théorème des restes chinois aux M_i, on obtient un entier m compris entre 0 et $n_1 * n_2 * n_3$ tel que $m = (\sum m_i)^3\ (mod\ n_1 *\ n_2 *\ n_3)$.

Car le message, $m_i \leq n_i$ donc $(\sum m_i)^3 < n_1 * n_2 * n_3$.
On a donc $m = (\sum m_i)^3$.

Exécution du code :

$Ensemble\ p : [37, 43, 61, 53, 59, 41]$
$Ensemble\ n : [1591, 3233, 2419]$
$n : 1591$
$m\ témoin : 1303$
$e : 3$
$Ensemble\ M : [993, 217, 1895]$
$m : 1303$

Exercice 14 : $A_4(n_1, n_2, n_3, e, M_1, M_2, M_3)$

Dans ce cas, nous savons que m contient au plus 3 bits dans sa représentation binaire.

Or, $m < n$ et $nombre\ de\ bits\ de\ m \leq\ nombre\ de\ bits\ de\ n$.

Donc dans le pire des cas, $m = 2^{nb\ bits\ de\ n}$.

Dans ce cas, il est possible de brute-force le message m en $O(n)$.

Exécution du code :

$Ensemble\ p : [37, 47, 43, 41, 59, 53]$
$Ensemble\ n : [1739, 1763, 3127]$
$n : 1739$
$m\ témoin\ binaire : 110000010\ -\ nb\ bits : 9$
$m\ témoin : 386$
$e : 859$
$Ensemble\ M : [490, 1160, 598]$
$m : 386$

Question subsidiaire : Variante de GenPremiers

Génération d'un nombre premier aléatoire de k bits en prenant en compte le produit des i premiers facteurs premiers.

Protocole expérimental :

Comme effectué précédemment (exo 6), on relève le temps d'exécution des deux versions de GenPremiers pour k allant de 1 à 16 et on compare le résultat.

Avant chaque exécution, on génère le produit des k premiers facteurs premiers.

Interprétation :

Etant donné que notre première version de GenPremiers s'appuie sur PrimaliteNaif, il est logique de voir que le temps d'exécution de la variante est inférieur sur long terme puisque GenPremiers(k, S) fait appel au test de Fermat. En effet, nous avons montré auparavant (exo 6) que le test de Fermat est asymptotiquement (dans le pire des cas) plus rapide que PrimaliteNaif.

Exécution du code :

Temps d'exécution en nanosecondes :
---------- Gen Premiers ----------
[0.0, 432900.0, 6200.0, 4000.0, 3100.0, 3700.0, 6700.0, 6200.0, 9700.0, 19200.0, 30600.0, 31600.0, 65000.0, 149100.0, 276700.0, 237100.0, 454100.0]
Total (en ms) : 1.7359ms
---------- Variante ----------
[0.0, 30400.0, 14700.0, 7800.0, 7400.0, 5200.0, 16600.0, 32900.0, 6800.0, 80900.0, 147200.0, 21000.0, 60800.0, 30400.0, 129500.0, 35500.0, 12500.0]
Total (en ms) : 0.6396ms