**Universidad de Granada**

*DEPARTAMENTO DE ÁLGEBRA*

Práctica 1 – Criptografía

**Aritmética Modular**

**Autores**: Alexander Moreno Borrego

Carlos Jesus Fernandez Basso

**Correos:** [alexmobo@correo.ugr.es](mailto:alexmobo@correo.ugr.es)

[mkarloos@correo.ugr.es](mailto:mkarloos@correo.ugr.es)

**DNI:** 39906263-K

**Grupo**: 3

**Profesor**: Jesús García Miranda

Contenido

[Especificación 3](#_Toc350939461)

[Descripción de los algoritmos 5](#_Toc350939462)

[Potenciación modular 5](#_Toc350939463)

[Logaritmo discreto 5](#_Toc350939464)

[Curvas elípticas 6](#_Toc350939465)

[Estudio de los tiempos 6](#_Toc350939466)

[Resutados potenciación modular 6](#_Toc350939467)

[Resultados logaritmo discreto 7](#_Toc350939468)

[Manual de uso de la aplicación 8](#_Toc350939469)

[SSOO Windows 8](#_Toc350939470)

[SSOO Linux o Mac 8](#_Toc350939471)

Especificación

Esta práctica nos introduce en algunos algoritmos de aritmética modular que son necesarios para el desarrollo de la criptografía de clave pública. Se trata de conocer e implementar algunos de ellos, y de observar las diferencias del tiempo de cálculo entre distintas operaciones. A lo largo de toda la práctica se utilizarán números naturales con un número elevado de cifras, por lo que será necesario optar por un lenguaje de programación que admita dichos cálculos, o bien adaptar, mediante el uso de bibliotecas específicas, el habitual para llevar a cabo estas tareas.

Para el desarrollo de la práctica es conveniente recordar el algoritmo extendido de Euclides, junto con su aplicación para calcular inversos modulares. No obstante, estos algoritmos no son objeto de la práctica.

Lo que hay que realizar es lo siguiente:

1. Implementar un programa que pida como entrada 3 números naturales (, y ) y dé como salida el número mod .
2. Implementar un programa que pida como entrada 3 números naturales (, y ) y decida si existe mod , y caso de que exista, debe dar una solución.

La solución puede no ser única. En tal caso, se puede optar por:

1. Dar una solución cualquiera.
2. Dar la menor solución.
3. Dar todas las soluciones menores que .

Para que el algoritmo funcione bien, el número p debe ser primo. En principio no hay que comprobar si, una vez introducidos los números, el tercer número es primo.

1. Realizar un estudio de los tiempos que emplea cada uno de los dos algoritmos anteriores. Para eso, tomamos números primos desde 5 cifras en adelante. Si es uno de esos números primos, entonces:
   1. Elegimos al azar dos números y , menores que , pero del mismo tamaño. Estos números podemos tomarlos nosotros. No es necesario utilizar una función que genere números aleatorios.
   2. Calculamos mod , y medimos el tiempo que tarda.
   3. Calculamos mod , y medimos el tiempo que tarda.

Si llegado un número primo para el que estamos haciendo la prueba el algoritmo tarda mucho, no es necesario seguir con el siguiente.

A continuación va una lista de posibles números primos.

46381

768479

9476407

36780481

562390847

1894083629

65398261921

364879542899

8590365927553

28564333765949

123456789101119

1. Resolver uno de los cuatro problemas siguientes:
   1. Factorización de números enteros.
   2. Cálculo de raíces cuadradas modulares.
   3. Sistemas de congruencias.
   4. Curvas elípticas.

Descripción de los algoritmos

Potenciación modular

El siguiente pseudocódigo es el que implementa la función de potenciación modular:

**función** pot\_mod2(a, b, m) **hacer**

x ← 1

**mientras** b > 0 **hacer**

**si** b%2 == 1 **hacer**

x ← (x\*a)%m

**fin si**

a ← (a\*a)%m

b ← b/2

**fin mientras**

**devolver** x

**fin función**

Logaritmo discreto

El pseudocódigo siguiente es el pseudocódigo que hemos diseñado para la función de logaritmo discreto.

**función** log\_dis(a, c, p) **hacer**

s ←

r ← Array vacío

t ← Array vacío

**para** i **desde** 0 **hasta** s-1 **hacer**

r.insertarPorDetras([, i])

t.insertarPorDetras([, i])

**fin para**

r ← ordenar(r)

t ← ordenar(t)

soluciones ← Array vacío

**para** **cada** **elemento** i **en** r **hasta**

j ← 0

**mientras** j <= i[0] YY j < len(t) **hacer**

**si** i[0] == t[j][0] **hacer**

solucion ← (s\*(i[1]+1))-t[j][1]

**si** solucion < p **hacer**

soluciones.insertarPorDetras(solucion)

**fin si**

j ← j + 1

**fin si**

**fin mientras**

**fin para**

**devolver** ordenar(soluciones)

**fin función**

Esta solución que hemos propuesto devuelve todas las soluciones inferiores a .

Curvas elípticas

Como el título indica hemos elegido realizar el problema de las curvas elípticas porque nos ha parecido realmente interesante ya que es el problema que se usa actualmente en el campo de la criptografía.

Estudio de los tiempos

Estos serán los números con los que probaremos la funcionalidad de las funciones descritos anteriormente.

|  |  |  |
| --- | --- | --- |
|  |  | o |
| 6540 | 7345 | 46381 |
| 74465 | 93624 | 768479 |
| 594637 | 662499 | 9476407 |
| 6347215 | 7694996 | 36780481 |
| 67754367 | 98548869 | 562390847 |
| 546552345 | 876889113 | 1894083629 |
| 2314567863 | 7658999244 | 65398261921 |
| 76898990876 | 99999999999 | 364879542899 |
| 347586977583 | 885455823590 | 8590365927553 |
| 3259874594584 | 7567458467455 | 28564333765949 |
| 75637476563633 | 97263447823703 | 123456789101119 |

Resutados potenciación modular

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | o | Resultado | T (s) |
| 6540 | 7345 | 46381 | 42382 | 0 |
| 74465 | 93624 | 768479 | 626960 | 0 |
| 594637 | 662499 | 9476407 | 9386511 | 0 |
| 6347215 | 7694996 | 36780481 | 12511599 | 0 |
| 67754367 | 98548869 | 562390847 | 220123032 | 0 |
| 546552345 | 876889113 | 1894083629 | 659348229 | 0 |
| 2314567863 | 7658999244 | 65398261921 | 5031006446 | 0 |
| 76898990876 | 99999999999 | 364879542899 | 189160340104 | 0 |
| 347586977583 | 885455823590 | 8590365927553 | 7706581485996 | 0 |
| 3259874594584 | 7567458467455 | 28564333765949 | 25493951987847 | 0 |
| 75637476563633 | 97263447823703 | 123456789101119 | 46007993118002 | 0 |

Resultados logaritmo discreto

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | o | Resultado | T (s) |
| 6540 | 7345 | 46381 | 27506 | 0,025 |
| 74465 | 93624 | 768479 | 105425 | 0,033 |
| 594637 | 662499 | 9476407 | - | 0,047 |
| 6347215 | 7694996 | 36780481 | - | 0,092 |
| 67754367 | 98548869 | 562390847 | 155172064 | 0,372 |
| 546552345 | 876889113 | 1894083629 | - | 0,702 |
| 2314567863 | 7658999244 | 65398261921 | 35335328857 | 8,757 |
| 76898990876 | 99999999999 | 364879542899 | 31067818263  213507589712 | 23,355 |
| 347586977583 | 885455823590 | 8590365927553 | - | 396,157 |
| 3259874594584 | 7567458467455 | 28564333765949 | - | - |
| 75637476563633 | 97263447823703 | 123456789101119 | - | - |

Las tablas anteriores muestran como la potenciación modular es una operación más sencilla de realizar que el cálculo del logaritmo discreto. Como curiosidad debemos decir que, al realizar el logaritmo discreto de los penúltimos números, el programa lanzó la excepción de *“Out of Mempory”* debido a la magnitud de los números con los que trabaja.

A partir de la sexta prueba es demasiado el tiempo que se tarda en obtener una solución al problema del cálculo del logaritmo discreto.

Podemos ver en la gráfica como el tiempo de resolución, si se puede resolver, del cálculo del logaritmo discreto obedece a una función exponencial.

Manual de uso de la aplicación

SSOO Windows

Para ejecutar el archivo ejecutable en Windows es tan sencillo como hacer doble clic sobre el archive .jar.

SSOO Linux o Mac

Para ejecutar el archive en Linux o Mac se necesita seguir una serie de pasos muy sencillos. Son los siguientes:

1. Abrir el terminal
2. Situarse en la carpeta donde tengamos el archivo ejecutable
3. Ejecutar la línea de comandos

*java –jar <nombre\_archivo>.jar*