Informe Tarea Algoritmos Probabilísticos

María Andrea Cruz Blandón
Edgar Andrés Moncada Taborda Luis Felipe Vargas Rojas

December 16, 2012

1 MonteCarlo

Basado en el teorema de Fermat extendido, algoritmo cuyo nombre es el de MIller-Rabin El cual se expresa en las diapositivas de la siguiente manera:

- Si n es un primo representado de la forma $n = 2^s * t + 1$
- Entonces si generamos un a tal que $a \in [2, n-2]$

Esta valor a debe cumplir que :

$a^t mod(n) == 1$ or $a^{2^i * t} mod(n) = n - 1$ para algún i tal que : $0 < i < s$

1.1 Implementación

El algoritmo se implemento con Java

Lo primero es preguntar si es par, si es así retorna no es primo.

Para conocer los valores s y t que representan la expresión del número primo en potencia de dos multiplicado por un factor se aplicaron las siguientes iteraciones.

```
long s=0;
long t= n-1;


while (ispar(t)){
        s++;
        t=t/2;
}
```

Como suponemos que n es primo y ya sabemos que no es par buscamos que t tome el primer par mas cercano osea $n-1$ empezamos a dividir entre dos de forma analoga aumentamos s que sera el exponente de 2 asi por cada división por dos sumamos iterativamente una potencia de dos. Finalmente encontramos los valores t y s que representan de la forma que necesitamos el valor.

Luego calculamos el nivel de certeza que queremos el cual define los k números aleatorios que se deben generar:

```
double e=certain;
double k = Math.log(1/e) / Math.log(2);

k= Math.ceil(k /2);
```

1.2 Problemas computacionales

El primer problema con el cual nos encontramos es el de expresar numeros o potencias tan grandes como 3450^{456} los cuales mediante el método que implementa la librería Math.pow nos encontrábamos con valores como infinito sin embargo existe un método llamado potencias modulares el cual encontramos en la siguiente fuente : http://www.slideshare.net/fivefingers/potencias-modulares

Que nos ayuda a definir valores de la forma $a^n mod(m)$ para a,n y m muy grandes la implementacion de esta idea fue :

```
calcularPotenciaModular(long base, long exponente, long modulo){


        long resultado = 1;
        for(int i = 1; i<=exponente; i++){
                resultado = (resultado*base)%modulo;

        }

        return resultado;
}
```

Ahora bien los limites de los cálculos están dados por el numero más grande que se puede representar con el tipo de dato long.

Una vez tenemos estas funciones finalmente el algoritmo quedo así:

```
        for (int j=0;j<k;j++) {

                long x= generador.aleatorio(number);
                if(!debeCumplir(x,t,s,number)) return false;

        }
```

Aqui generamos un numero aleatorio $x \in [2, number - 2]$ Lo generamos k veces donde k es determinado por el nivel de certeza. Donde debeCumplir es una función que implementa las condiciones mencionadas al inicio y se codifico así:

```
public boolean debeCumplir(long a,long t , long s, long number){


long x=a;


long modulo = calcularPotenciaModular(x,t,number);
```

```
if (modulo==1) {


        return true;}
else {


        for (int i=0;i<=s;i++){
                int  potencia = (int) ( Math.pow(2,i)*t);

                long modulo2 = calcularPotenciaModular(x,potencia,number);



                if(modulo2 == number-1    ){


                        return true;

                }

        }


}



return false ;




}
```

Ya con este algoritmo implementado podemos definir con cierta probabilidad si un numero es primo o no.

2 Crear Dos Primos Muy Grandes

Para esta implementación se hizo lo siguiente:

```
public void buscarPrimoGrande(){


Generador generadorGrandes= new Generador(543);
long  primo1=0,primo2 = 0;
```

```
while(true){
        long mayorDeMillon=generadorGrandes.aleatorioMillon();

        if (MillerRabinPrimalityTest(0.001, mayorDeMillon ))
        {
                primo1=mayorDeMillon;
                break;
        }
}

while(true){
        long mayorDeMillon=generadorGrandes.aleatorioMillon();

        if (primo1!=primo2 && MillerRabinPrimalityTest(0.001, mayorDeMillon ))
        {

                primo2=mayorDeMillon;
                break;

        }
}

}
```

Basicamente creamos un bucle indefinido que genera numeros aleatorios mayores de un millon en cada iteracion preguntamos si es primo si no sigue generando hasta encontrarlo cuando lo encuentre se detiene y busca el otro primo que debe ser diferente al primero.

3 Simulación de Proceso de Envió de Mensajes Cifrados

Para este parte nos valimos de un chat implementado en Java a través sockets.

Si B quiere recibir un mensaje genera dos primos muy grandes con la clase explicada anteriormente definimos z igual al producto de los primos muy grandes f = (primo1-1)* (primo2-1). Buscamos un n que no tenga factores comunes con f que a su ves exista un único s tal que ns mod f = 1.

A recibe n y z b se queda con s que es único.

A escribe el mensaje vuelve su mensaje numero a = ASCII del mensaje . si a ¡ z hace lo siguiente $c = a^n mod(z)$

si no es así el mensaje debe ser dividido en varios mensajes y mandara el c a B.

4 Complejidad del proceso para una persona que intente romper el código.

Si una tercera persona contara con los valores de n y z públicos averiguar el valor de s tal que pueda decodificar el mensaje encriptado, será tan complejo como grandes sean los números primos elegidos. Pues en primera instancia cotejar que valor de s permite ns mod z = 1 será en función del tamaño de los valores de n y z y como bien sabemos estos dependen fuertemente de los primos elegidos. Una vez se tenga el valor de s la persona "hacker" podrá descifrar el mensaje.

Sin embargo a este proceso de búsqueda le afecta un limite y éste es el computacional, puesto que como sabemos buscar si un numero es primo es un problema de decisión NP. Cuanto mas grande hallan sido los primos elegidos para el proceso de cifrado mucho mas tiempo y recursos necesitará la máquina para descubrirlos.