PUZLES HASH

Práctica 5

Índice

Funciones

1. Constructor

```java
public class Block {

    byte[] id;
    byte[] x;
    byte[] hash;
    int nIntentos = 0;
    byte[] textBytes;
    int b;
    byte[] inputBitsChain;
    boolean increase;

    public Block(String text, int b, byte[] inputBytes, boolean increase) {
        this.textBytes = text.getBytes(StandardCharsets.UTF_8); //Pasamos el texto a bits
        this.b = b;
        this.inputBitsChain = inputBytes;
        this.increase = increase;

        create();

    }
```

Al constructor de la clase se le pasa un texto, el número de bits que tienen que ser 0 (b), una cadena de bits y un booleano que indica si la cadena de bits que se uno al id será incremental o aleatoria.

Además, se guardan los valores en variables de clase, lo que facilitará el desarrollo de la práctica y evitará tener que reescribir el mismo valor una y otra vez.

El texto se pasa a un array de bytes.

2. Función crear

```java
    private void create() {

        MessageDigest sha256 = null;

        try {
            sha256 = MessageDigest.getInstance("SHA-256");
        } catch (NoSuchAlgorithmException e1) {
            e1.printStackTrace();
            System.err.println("El algoritmo de hash no se enuentra.");
        }

        //creamos el id
        id = concatBytesArray(textBytes, inputBitsChain);

        //generamos una cadena aleatoria
        byte[] randomChanin = genRandomChanin(inputBitsChain.length);

        //Le añadimos al id una cadena aleatoria del tamaño n
        x = concatBytesArray(id, randomChanin);

        //Hacemos el hash
        hash = sha256.digest(x);

        nIntentos = 0;

        while(!checkBits(hash, b)) { //Comprobamos los bits a 0
            if(increase)
                x = concatBytesArray(id, increaseChanin(randomChanin)); // Función aleatoria
            else
                x = concatBytesArray(id, genRandomChanin(inputBitsChain.length)); // Función incremental

            //Hacemos el hash
            hash = sha256.digest(x);
            // Aumentamos el número de intentos
            nIntentos++;
        }
    }
```

Esta es la función principal, en ella es donde se crean los bloques con los parámetros que nos han pasado en el constructor.

Las operaciones que realizan son las siguientes:

1. Lo primero que se hace es crear un objeto hash, en este caso un sha256.
2. Creamos la id como una concatenación de los bits del mensaje y de los bits que nos han pasado en el constructor.
3. Generamos una secuencia aleatoria de bits, mediante la función "*genRandomChanin*".
4. Creamos una secuencia de bits como concatenación de la id y la secuencia anterior.
5. A la secuencia anterior le generamos el hash mediante el objeto hash creado en el punto 1.
6. Comprobamos si el número de bits a 0 es mayor o igual que el que nos han indicado en el constructor en la variable b. Esto se hace mediante la función "*checkBits*"
7. Si no es igual, entramos en el while y repetiremos los pasos del 8 al 11 hasta que el número de 0s que salen del hash sea correcto.
8. Generamos una nueva secuencia de bits, en este caso esta puede ser aleatoria o incremental, en función de lo que se le haya pasado al constructor.
   Si es aleatoria se llamará a la función "*genRandomChanin*" y si es incremental a la función "*increaseChanin*".
9. Volvemos a calcular el hash de la concatenación del id con una de las secuencias anteriores.
10. Aumentamos en 1 el número de intentos.
11. Y volvemos a comprobar si el resultado es correcto.

3. Función concatBytesArray

```
125    private byte[] concatBytesArray(byte[] arr1, byte[] arr2) {
126        byte[] ret = new byte[arr1.length + arr2.length];
127
128        System.arraycopy(arr1, 0, ret, 0, arr1.length);
129        System.arraycopy(arr2, 0, ret, arr1.length, arr2.length);
130
131        return ret;
132    }
133
134    private String toStringBytesArray(byte[] arr) {
135        String out = "";
136        for(byte b:arr)
137            out +=  "" + b + " ";
138
139        return out;
140    }
```

A esta función le llegan dos arrays de byte y los concatena, devolviendo un único array como concatenación de los otros dos.

4. Función genRandomChanin

```
85    private byte[] genRandomChanin(int n) {
86
87        byte[] rantomChanin = new byte[n];
88
89        Random r = new Random();
90        for(int i = 0; i < n; i++) {
91            rantomChanin[i] = (byte) r.nextInt();
92        }
93
94        return rantomChanin;
95    }
```

Esta función genera una secuencia aleatoria de bits, para ello se juega con que el tipo byte en java es un entero con signo de 8 bits, por lo que a cada bloque se le asigna un número aleatorio y se devuelve un array de bytes del tamaño dado.

5. Función increaseChanin

```
67    static byte[] increaseChanin(byte[] chain) {
68
69        int i = 0;
70        while(i < chain.length && chain[i] == -1){i++;}
71        if(i < chain.length) {
72            chain[i]++;
73            if(i > 0 && (chain[i]%2 == 0 || chain[i] == 1))
74                for(int j = 0; j < i; j++)
75                    chain[j] = 0;
76        }else {
77            for(i = 0; i < chain.length; i++)
78                chain[i] = 0;;
79        }
80
81        return chain;
82
83    }
```

Esta función lo que hace es incrementar en uno la secuencia de bits aleatorios.

Para hacer esto:

1. Mira el primer bloque que tenga algún bit a 0.
2. Si lo encuentra, incrementa el valor del bloque, y si no, reinicia el array a 0, ya que tenía todos los valores a 1.
3. Si al incrementarlo, el valor del bloque es 1 o múltiplo de 2, reinicia a 0 el valor de los bloques anteriores.

6. Función checkBits

```
97    static boolean checkBits(byte[] bytes, int n) {
98
99        boolean salida = true;
100       // Si n > 8, los primeros bits tienen que estar a 0 ==> bytes[i] == 0
101       int i;
102       for(i = 0; i < n / 8 && salida; i++) {
103           salida = bytes[i] == 0;
104       }
105
106       // Si salida sigue siendo true
107       if(salida) {
108           n = n % 8;
109
110           // para los que nos queden
111           if(n > 0 && bytes[i] >= 0) {
112               salida = bytes[i] < Math.pow(2, 8 - n);
113           }else {// Si es negativo, el numero de bits de ese bloque tiene que ser 0
114               salida = n == 0;
115           }
116
117       }
118
119       return salida;
120   }
```

Esta función nos indicará si dada una secuencia de bits, los n primeros son 0 o no.

Para esto, sigue los siguientes pasos:

1. Si n es mayor que 8, comprobamos que los primeros bloques de 8 bits valen 0, es decir, tienen todos los bits a 0.
2. Si esta condición se cumple, miramos el bloque que ya no tiene que tener todos los bits a 0, solo los m primeros siendo m < 8, para esto lo primero que hacemos es n = n % 8, línea 108.
3. Llegados a este punto, hay tres opciones:
    a. Que n sea 0, en tal caso da igual el valor del bloque, por lo que salaríamos por el else con salida = true.
    b. Que n sea > 0 y el bloque tenga un valor negativo, en tal caso volvemos a salir por el else con salida = false, ya que el primer bit, el del signo estaría a 1 y no cumpliría la condición.
    c. Que n sea > 0 y el bloque sea mayor o igual a 0, en este caso hay que comprobar el valor del bloque y jugando por cómo se almacenan los datos en función del valor, se puede determinar la posición del primer cero.

Valor en decimal 2^{8-n} con n ϵ (0,8)	Valor en binario	Si el valor es menor, el primer 1 está detrás de la posición
$128 = 2^7$	1000 0000	1
$64 = 2^6$	0100 0000	2
$32 = 2^5$	0010 0000	3
$16 = 2^4$	0001 0000	4
$8 = 2^3$	0000 1000	5
$4 = 2^2$	0000 0100	6
$2 = 2^1$	0000 0010	7
$1 = 2^0$	0000 0001	El valor es 0.

7. Función nextBlock

```
139     public Block nextBlock() {
140         inputBitsChain = hash;
141         create();
142         return this;
143     }
```

Esta función se llama para recalcular el siguiente bloque, para ello, se iguala la entrada de bits al hash y se llama de nuevo a la función create(), de modo que al calcular el siguiente bloque, los bits de entrada sean los bits del hash del bloque anterior.

Ejecución

```
42     private static void blockChain(int bParam, int repeticiones, boolean increase) {
43
44         for(int b = 0; b <= bParam; b ++) {
45             //Generamos el primer bloque, siempre con mi nombre como texto
46             block = new Block("Pedro Luis Fuertes Moreno", b, bi, increase);
47             //Iniciamos la variable para cada b, con el número de intentos para generar el primer bloque
48             int nintentos = block.getnIntentos();
49             for(int i = 1; i < repeticiones; i++) {
50                 block.nextBlock(); //Llamamos a nextBlock 9 veces más
51                 nintentos += block.getnIntentos(); // Guardamos el número de intentos para generar cada bloque
52
53             }
54             //System.out.println(block);
55             System.out.println(nintentos/repeticiones + "\t" + b); // Calculamos la media y la imprimimos.
56         }
57     }
58
```

Para hacer las pruebas, se genera un primer bloque con los parámetros correspondientes, como entrada de texto mi nombre, la b que empieza en 0 y llega al número de repeticiones que queramos y un boolean que nos diferencia entre una función u otra, es decir, si queremos que la cadena aleatoria se incremente en cada iteración o que se genere una nueva.

Inicializamos una variable entera con el número de intentos del primer bloque, la cual luego se incrementará con el número de intentos para generar cada uno de los bloques, esto nos servirá para calcular la media.

A continuación, entramos en un bucle para generar el número de bloques con dichos parámetros (texto de entrada, el número de bits que tienen que estar a 0, la b y si la variable aleatoria de incrementa o se genera nueva) que nos hayan indicado en el parámetro "*repeticiones*" y en cada repetición, sumamos el número de intentos hasta encontrar la secuencia que hace que la función hash tenga "*b*" ceros al principio, en la variable "*nIntentos*".

Tras encontrar todos los bloques de la secuencia, nos imprime la media.

Por tanto, en el main queda como sigue:

```
60     public static void main(String[] args) {
61         Random random = new Random();
62         for(int i = 0;i < randomInput.length; i ++) {
63             randomInput[i] = (byte) random.nextInt();
64         }
65
66         int b = 22;
67
68         System.out.println("Aleatorio");
69         blockChain(b, 10, false);
70         System.out.println("\nIncremental");
71         blockChain(b, 10, true);
72
73     }
```

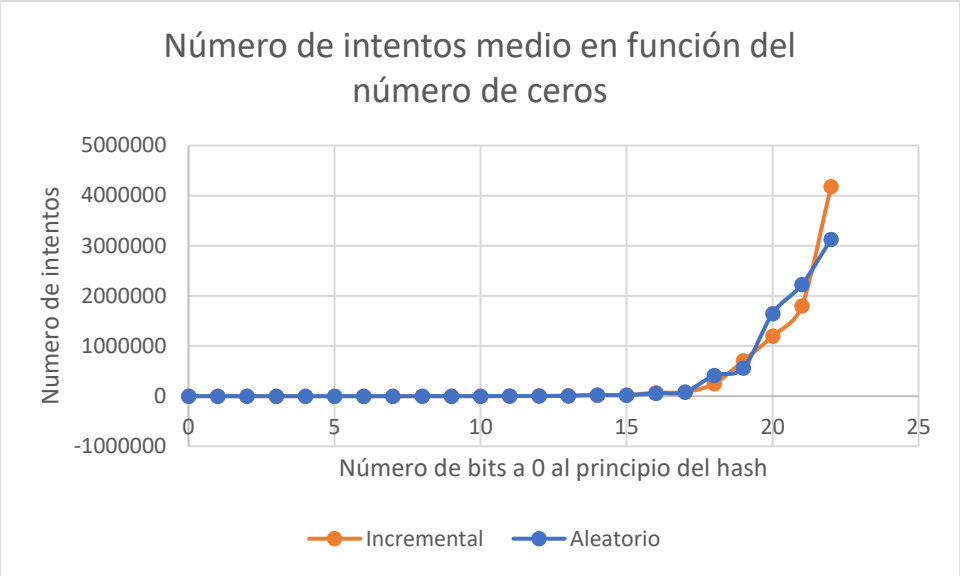
Generamos una secuencia aleatoria, la cual compartirán todos los bloques, esto evitará cierta incertidumbre.

Fijamos la b máxima que queremos alcanzar, y llamamos dos veces a la función blockChain, una con la variable aleatoria incremental y otra vez cambiándola.

De esta manera ya tendríamos la secuencia completa.

Tabla

b	Aleatorio	Incremental
0	0	0
1	0	0
2	3	2
3	6	3
4	10	16
5	34	41
6	69	66
7	218	112
8	187	276
9	678	260
10	774	824
11	2922	1138
12	4340	2812
13	7462	9256
14	23901	19117
15	19545	24943
16	57795	68626
17	83955	83930
18	416715	249007
19	562684	707271
20	1647459	1201918
21	2225130	1799201
22	3128496	4178871



Conclusiones

Tal y como puede verse en la gráfica, a medida que aumenta el número de ceros que se quieren conseguir al hacer el hash, el número de intentos necesarios para dar con la secuencia que genere dicho hash aumenta de manera exponencial, por lo que conseguir una secuencia que de una gran cantidad de bits a cero, es una tarea realmente complicada.

Esta dificultad para obtener dichas secuencias puede hacer que dichas secuencias tengan “valor” y sirvan como moneda, este es el principio de las criptomonedas.