
Abner Josué Miguel Xocop Chacach
20180515
Ingeniería en Computer Science

**Hash Functions**

Curso: Estructura de Datos
Catedrático: Fernandojose Boiton
Área: Computer Science

Abril de 2019

# Índice

**Funciones Hash**

## BLAKE y BLAKE2

BLAKE y BLAKE2 son funciones hash criptográficas basado en Dan Bernstein 's ChaCha cifrado de flujo, pero una copia permutado del bloque de entrada, XORed con algunas constantes redondas, se añaden antes de cada ronda ChaCha. Al igual que SHA-2, hay dos variantes que se diferencian en el tamaño de la palabra. ChaCha opera en una matriz de palabras 4 × 4. BLAKE combina repetidamente un valor hash de 8 palabras con 16 palabras de mensaje, truncando el resultado de ChaCha para obtener el siguiente valor hash. BLAKE-256 y BLAKE-224 usan palabras de 32 bits y producen tamaños de resumen de 256 bits y 224 bits, respectivamente, mientras que BLAKE-512y BLAKE-384 usa palabras de 64 bits y produce tamaños de resumen de 512 bits y 384 bits, respectivamente.

BLAKE2b es más rápido que SHA-3, SHA-2, SHA-1 y MD5 en arquitecturas de 64 bits x64 y ARM. BLAKE2 proporciona seguridad superior a SHA-2 y similar a la de SHA-3: inmunidad a la extensión de longitud, indiferenciabilidad de un oráculo aleatorio, etc.

Algoritmo

Al igual que SHA-2, BLAKE viene en dos variantes: una que usa palabras de 32 bits, que se usa para calcular hashes de hasta 256 bits de longitud, y otra que usa palabras de 64 bits, que se usa para calcular hashes de hasta 512 bits de longitud. La transformación del bloque central combina 16 palabras de entrada con 16 variables de trabajo, pero solo 8 palabras (256 o 512 bits) se conservan entre los bloques.

La operación principal, equivalente a la cuarta ronda de ChaCha, opera en una columna o diagonal de 4 palabras a b c d, que se combina con 2 palabras de mensaje m[]y dos palabras constantes n[]. Se realiza 8 veces por ronda completa:

```
1.  j ← σ [r% 10] [2 × i]          // Cálculos del índice
2.  k ← σ [r% 10] [2 × i + 1]
3.  a ← a + b + (m [j] ⊕ n [k])    / / Paso 1 (con entrada)
4.  d ← (d ⊕ a) >>> 16
5.  c ← c + d                      // Paso 2 (sin entrada)
6.  b ← (b ⊕ c) >>> 12
7.  a ← a + b + (m [k] ⊕ n [j])    // Paso 3 (con entrada)
8.  d ← (d ⊕ a) >>> 8
9.  c ← c + d                      // Paso 4 (sin entrada)
10. b ← (b ⊕ c) >> > 7
```

En lo anterior, r el número redondo (0–13), y varía de 0 a 7.

Algoritmo Blake2

```
1.  Algorithm BLAKE2b
```

```
2.    Input:
3.      M                       Message to be hashed
4.      cbMessageLen: Number, (0..2128)  Length of the message in bytes
5.      Key                     Optional 0..64 byte key
6.      cbKeyLen: Number, (0..64)       Length of optional key in bytes
7.      cbHashLen: Number, (1..64)      Desired hash length in bytes
8.    Output:
9.      Hash                    Hash of cbHashLen bytes
10.
11.   Initialize State vector h with IV
12.   h0..7 ← IV0..7
13.
14.   Mix key size (cbKeyLen) and desired hash length (cbHashLen) into h0
15.   h0 ← h0 xor 0x0101kknn
16.       where kk is Key Length (in bytes)
17.            nn is Desired Hash Length (in bytes)
18.
19.   Each time we Compress we record how many bytes have been
      compressed
20.   cBytesCompressed ← 0
21.   cBytesRemaining  ← cbMessageLen
22.
23.   If there was a key supplied (i.e. cbKeyLen > 0)
24.   then pad with trailing zeros to make it 128-bytes (i.e. 16 words)
25.   and prepend it to the message M
26.   if (cbKeyLen > 0) then
27.     M ← Pad(Key, 128) || M
28.     cBytesRemaining ← cBytesRemaining + 128
29.   end if
30.
31.   Compress whole 128-byte chunks of the message, except the last chunk
32.   while (cBytesRemaining > 128) do
33.     chunk ← get next 128 bytes of message M
34.     cBytesCompressed ← cBytesCompressed + 128  increase count of bytes
      that have been compressed
35.     cBytesRemaining  ← cBytesRemaining  - 128  decrease count of bytes in
      M remaining to be processed
36.
37.     h ← Compress(h, chunk, cBytesCompressed, false)  false ⇒ this is not the
      last chunk
38.   end while
39.
40.   Compress the final bytes from M
41.   chunk ← get next 128 bytes of message M  We will get cBytesRemaining
      bytes (i.e. 0..128 bytes)
42.   cBytesCompressed ← cBytesCompressed+cBytesRemaining  The actual
      number of bytes leftover in M
```

```
43.   chunk ← Pad(chunk, 128)  If M was empty, then we will still compress a
      final chunk of zeros
44.
45.   h ← Compress(h, chunk, cBytesCompressed, true)  true ⇒ this is the last
      chunk
46.
47.   Result ← first cbHashLen bytes of little endian state vector h
48. End Algorithm BLAKE2b
```

Ejecución

El tiempo de ejecución de esta función es de O(m).

Ventajas

BLAKE2 admite los modos de codificación, salado, personalización y hash tree, y puede generar resúmenes de 1 a 64 bytes para BLAKE2b o hasta 32 bytes para BLAKE2s. También hay versiones paralelas diseñadas para un mayor rendimiento en procesadores multi-core; BLAKE2bp (paralelo de 4 vías) y BLAKE2sp (paralelo de 8 vías).

Hay una variante de "Función de salida extensible" (XOF) de BLAKE2 llamada "BLAKE2X", que puede generar una cantidad muy grande de bits aleatorios (en lugar de solo 256 o 512).

## Algoritmo de Luhn

También conocido como algoritmo "módulo o 10" o "mod 10", llamado así por su creador el científico de IBM Hans Peter Luhn. Consiste en una simple suma de comprobación de fórmula utilizada para validar una variedad de números de identificación, tales como crédito, números de tarjetas, números de IMEI, números de identificador de proveedor de nacionales en los Estados Unidos y los códigos que aparecen en la factura para validar la encuesta de McDonalds, Taco Bell, y Tractor Supply Co.

Fue diseñado para proteger contra errores accidentales, no ataques maliciosos. La mayoría de las tarjetas de crédito y muchos números de identificación del gobierno utilizan el algoritmo como un método simple para distinguir números válidos de números mal escritos o incorrectos.

Descripción

La fórmula verifica un número contra su dígito de verificación incluido, que generalmente se agrega a un número de cuenta parcial para generar el número de cuenta completo. Este número debe pasar la siguiente prueba:

- Desde el dígito más a la derecha, que es el dígito de control, y moviéndose a la izquierda, doble el valor de cada segundo dígito. El dígito de control

no se duplica; el primer dígito duplicado está inmediatamente a la izquierda del dígito de control. Si el resultado de esta operación de duplicación es mayor que 9 (por ejemplo, 8 × 2 = 16), agregue los dígitos del resultado (por ejemplo, 16: 1 + 6 = 7, 18: 1 + 8 = 9) o, alternativamente, el mismo resultado final se puede encontrar restando 9 de ese resultado (por ejemplo, 16: 16 - 9 = 7, 18: 18 - 9 = 9).
- Toma la suma de todos los dígitos.
- Si el módulo total 10 es igual a 0 (si el total termina en cero), entonces el número es válido de acuerdo con la fórmula de Luhn; de lo contrario no es válido.

Supongamos un ejemplo de un número de cuenta "7992739871" al que se le agregará un dígito de control, con el formato 7992739871x:

| **Account number** | 7 | 9 | 9 | 2 | 7 | 3 | 9 | 8 | 7 | 1 | **x** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Double every other** | 7 | **18** | 9 | **4** | 7 | **6** | 9 | **16** | 7 | **2** | **x** |
| **Sum digits** | 7 | **9** | 9 | 4 | 7 | 6 | 9 | **7** | 7 | 2 | **x** |

La suma de todos los dígitos en la tercera fila es 67 + x.

El dígito de control (x) se obtiene calculando la suma de los dígitos que no son de control y luego computando 9 veces ese valor módulo 10 (en forma de ecuación, ((67 × 9) mod 10)). En forma de algoritmo:

- Calcule la suma de los dígitos sin verificación (67).
- Multiplica por 9 (603).
- El dígito de las unidades (3) es el dígito de control. Por lo tanto, x = 3.

Ventajas y desventajas

- Ejecución en O(1).
- El algoritmo de Luhn detectará cualquier error de un solo dígito, así como casi todas las transposiciones de dígitos adyacentes. Sin embargo, no detectará la transposición de la secuencia de dos dígitos 09 a 90 (o viceversa). Detectará 7 de los 10 errores gemelos posibles (no detectará 22 ↔ 55, 33 ↔ 66 o 44 ↔ 77).
- Otros algoritmos de comprobación de dígitos más complejos (como el algoritmo Verhoeff y el algoritmo Damm) pueden detectar más errores de transcripción. El algoritmo Luhn mod N es una extensión que admite cadenas no numéricas.
- Los sistemas que se ajustan a un número específico de dígitos (por ejemplo, mediante la conversión de 1234 a 0001234) pueden realizar la validación de Luhn antes o después del relleno y lograr el mismo resultado.

Pseudocódigo

```
1.  función checkLuhn (string purportedCC) {
2.      int sum: = integer (purportedCC [length (purportedCC) -1])
3.      int nDigits: = length (purportedCC)
4.      int parity: = nDigits modulus 2
5.      para i de 0 a nDigits - 2 {
6.          int digit: = número entero (pretendido CC [i])
7.          si i módulo 2 =
8.              dígito de paridad : = dígito × 2
9.          si dígito> 9
10.             dígitos: = dígito - 9
11.         suma: = suma + dígito
12.     }
13.     retorno (módulo de suma 10) = 0
14. }
```

## BSD Checksum

Es un algoritmo de suma de comprobación heredado y de uso común. Se ha implementado en BSD y también está disponible a través de la utilidad de línea de comandos de suma de GNU.

Cálculo de la suma de comprobación BSD

Calcula una suma de comprobación de 16 bits sumando todos los bytes (palabras de 8 bits) del flujo de datos de entrada. Con el fin de evitar muchas de las debilidades de simplemente agregar los datos, el acumulador de suma de comprobación se gira circularmente hacia la derecha un bit en cada paso antes de agregar el nuevo char.

```
1.  int  bsdChecksumFromFile ( FILE  * fp )
2.  / * El identificador de archivo para los datos de entrada * /
3.  {
4.       int  checksum  =  0 ;
5.  / * La suma de verificación mod 2 ^ 16. * /
6.  for  ( int  ch  =  getc ( fp );
7.  ch  ! =  EOF ;
8.  ch  =  getc ( fp ))
9.  {
10. checksum  =  ( checksum  >>  1 )  +  (( checksum  &  1 )  15 );
11.   suma de control  + =  ch ;
12.         suma de control  & =  0xffff ;
13.  / * Mantenerlo dentro de los límites. * /
14. }
15.  suma de comprobación de retorno;
16. }
```

este algoritmo calcula una suma de comprobación al segmentar los datos y al agregarlos a un acumulador que se desplaza a la derecha circular entre cada suma. Para mantener el acumulador dentro de los límites del valor de retorno, se realiza el enmascaramiento de bits con 1's.

Ejemplo: Calcular una suma de comprobación de 4 bits utilizando segmentos de tamaño de 4 bits (big-endian)

Entrada: 101110001110 -> tres segmentos: 1011, 1000, 1110.

Iteración 1:

segmento: 1011 suma de comprobación: 0000 máscara de bits: 1111

a) Aplicar desplazamiento circular a la suma de control:

0000 -> 0000

b) Agregue la suma de comprobación y el segmento juntos, aplique la máscara de bits al resultado obtenido:

0000 + 1011 = 1011 -> 1011 y 1111 = 1011

Iteración 2:

segmento: 1000 suma de comprobación: 1011 máscara de bits: 1111

a) Aplicar desplazamiento circular a la suma de control:

1011 -> 1101

b) Agregue la suma de comprobación y el segmento juntos, aplique la máscara de bits al resultado obtenido:

1101 + 1000 = 10101 -> 10101 & 1111 = 0101

Iteración 3:

segmento: 1110 suma de comprobación: 0101 máscara de bits: 1111

a) Aplicar desplazamiento circular a la suma de control:

0101 -> 1010

b) Agregue la suma de comprobación y el segmento juntos, aplique la máscara de bits al resultado obtenido:

1010 + 1110 = 11000 -> 11000 y 1111 = 1000

Suma de control final: 1000

## Algoritmo de Verhoeff

El algoritmo de Verhoeff es una fórmula de suma de comprobación para la detección de errores desarrollada por el matemático holandés Jacobus Verhoeff y se publicó por primera vez en 1969. Fue el primer algoritmo de dígito de verificación decimal que detecta todos los errores de un solo dígito, y todos los errores de transposición que involucran dos dígitos adyacentes, que en ese momento se creían imposibles con dicho código.

Descripción

Verhoeff ideó su algoritmo usando las propiedades del grupo diédrico de orden 10 (un sistema no conmutativo de operaciones en diez elementos, que corresponde a la rotación y reflexión de un pentágono regular), combinado con una permutación. Afirmó que era el primer uso práctico del grupo diédrico y confirmó el principio de que al final, todas las matemáticas encontrarán un uso, aunque en la práctica el algoritmo se implementará utilizando tablas de búsqueda simples sin necesidad de comprender cómo generar esas tablas desde el grupo subyacente y la teoría de la permutación.

Esto se considera más adecuadamente como una familia de algoritmos, ya que hay otras permutaciones posibles, y se analizan en el tratamiento de Verhoeff. Él señala que esta permutación particular,

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 5 & 7 & 6 & 2 & 8 & 3 & 0 & 9 & 4 \end{pmatrix} = (1 \quad 5 \quad 8 \quad 9 \quad 4 \quad 2 \quad 7 \quad 0)(3 \quad 6)$$

Es especial ya que tiene la propiedad de detectar el 95.3% de los errores fonéticos.

Ventajas y desventajas

- Las fortalezas del algoritmo son que detecta todos los errores de transliteración y transposición, y, además, la mayoría de los gemelos, saltos gemelos, transposiciones de saltos y errores fonéticos.
- La principal debilidad del algoritmo de Verhoeff es su complejidad. Los cálculos requeridos no pueden expresarse fácilmente como una fórmula. Las tablas de búsqueda son necesarias para un cálculo fácil. Un código similar es el algoritmo de Damm, que tiene cualidades similares.

Algoritmo basado en la tabla

El algoritmo de Verhoeff se puede implementar utilizando tres tablas: una tabla de multiplicar d, una tabla inversa inv, y una tabla de permutación p.

| $d(j,k)$ [8] | | k | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| j | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | 1 | 1 | 2 | 3 | 4 | 0 | 6 | 7 | 8 | 9 | 5 |
| | 2 | 2 | 3 | 4 | 0 | 1 | 7 | 8 | 9 | 5 | 6 |
| | 3 | 3 | 4 | 0 | 1 | 2 | 8 | 9 | 5 | 6 | 7 |
| | 4 | 4 | 0 | 1 | 2 | 3 | 9 | 5 | 6 | 7 | 8 |
| | 5 | 5 | 9 | 8 | 7 | 6 | 0 | 4 | 3 | 2 | 1 |
| | 6 | 6 | 5 | 9 | 8 | 7 | 1 | 0 | 4 | 3 | 2 |
| | 7 | 7 | 6 | 5 | 9 | 8 | 2 | 1 | 0 | 4 | 3 |
| | 8 | 8 | 7 | 6 | 5 | 9 | 3 | 2 | 1 | 0 | 4 |
| | 9 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | $inv(j)$ |
|---|---|---|
| j | 0 | 0 |
| | 1 | 4 |
| | 2 | 3 |
| | 3 | 2 |
| | 4 | 1 |
| | 5 | 5 |
| | 6 | 6 |
| | 7 | 7 |
| | 8 | 8 |
| | 9 | 9 |

| $p(pos,num)$ | | una | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| $pos \pmod 8$ | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| | 1 | 1 | 5 | 7 | 6 | 2 | 8 | 3 | 0 | 9 | 4 |
| | 2 | 5 | 8 | 0 | 3 | 7 | 9 | 6 | 1 | 4 | 2 |
| | 3 | 8 | 9 | 1 | 6 | 0 | 4 | 3 | 5 | 2 | 7 |
| | 4 | 9 | 4 | 5 | 3 | 1 | 2 | 6 | 8 | 7 | 0 |
| | 5 | 4 | 2 | 8 | 6 | 5 | 7 | 3 | 9 | 0 | 1 |
| | 6 | 2 | 7 | 9 | 3 | 8 | 0 | 6 | 4 | 1 | 5 |
| | 7 | 7 | 0 | 4 | 6 | 9 | 1 | 3 | 2 | 5 | 8 |

La primera tabla, d, se basa en la multiplicación en el grupo diedro $D_5$ y es simplemente la tabla de Cayley del grupo. Este grupo no es conmutativo, es decir, para algunos valores de j y k, d (j, k) ≠ d (k, j).

La tabla inversa inv representa el inverso multiplicativo de un dígito, es decir, el valor que satisface d (j, inv (j)) = 0.

La tabla de permutación p aplica una permutación a cada dígito en función de su posición en el número. Esto es en realidad una única permutación (1 5 8 9 4 2 7 0) (3 6) aplicada de forma iterativa; es decir, p (i + j, n) = p (i, p (j, n)).

El cálculo de la suma de comprobación de Verhoeff se realiza de la siguiente manera:

- Cree una matriz n a partir de los dígitos individuales del número, tomada de derecha a izquierda (el dígito más a la derecha es n 0, etc.).
- Inicialice la suma de comprobación c a cero.
- Para cada índice i de la matriz n, comenzando en cero, reemplace c con d (c, p (i mod 8, $n_i$)).
- El número original es válido si y solo si c = 0.

Para generar un dígito de control, agregue un 0, realice el cálculo: el dígito de control correcto es inv (c).

Ejemplos

**Generar** un dígito de control para 236:

| yo | $n_i$ | $p(y, n_y)$ | c |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 6 | 3 | 3 |
| 2 | 3 | 3 | 1 |
| 3 | 2 | 1 | 2 |

c es 2, por lo que el dígito de control es inv (2), que es 3.

Valide el dígito de control 2363:

| yo | $n_i$ | $p(y, n_y)$ | c |
|---|---|---|---|
| 0 | 3 | 3 | 3 |
| 1 | 6 | 3 | 1 |
| 2 | 3 | 3 | 4 |
| 3 | 2 | 1 | 0 |

c es cero, por lo que la comprobación es correcta.

## Algoritmo DAMM

Es un check digit algorithm que detecta todos los errores de un solo dígito y todos los errores de transposición adyacentes. Fue presentado por H. Michael Damm en 2004.

Fortalezas

- El algoritmo de Damm es similar al algoritmo de Verhoeff. También detectará todas las apariciones de los dos tipos de errores de transcripción que aparecen con mayor frecuencia, a saber, la alteración de un solo dígito y la transposición de dos dígitos adyacentes (incluida la transposición del dígito de control final y el dígito anterior). Pero el algoritmo de Damm tiene el beneficio que ofrece sin las permutaciones construidas específicamente y sus poderes específicos de posición son inherentes al esquema de Verhoeff. Además, se puede prescindir de una tabla de inversos si todas las entradas diagonales principales de la tabla de operaciones son cero.
- El algoritmo de Damm no sufre de exceder el número de 10 valores posibles, lo que hace que sea necesario utilizar un carácter sin dígitos.
- La anticipación de los ceros a la izquierda no afecta el dígito de control.
- Existen cuasigrupos totalmente anti-simétricos que detectan todos los errores fonéticos asociados con el idioma inglés (13 ↔ 30, 14 ↔ 40, ..., 19 ↔ 90). La tabla utilizada en el ejemplo ilustrativo se basa en una instancia de este tipo.

Desventajas

A pesar de sus propiedades deseables en contextos típicos donde se utilizan algoritmos similares, el algoritmo de Damm es en gran parte desconocido y poco utilizado en la práctica.

Dado que la adición de ceros a la izquierda no afecta al dígito de verificación, los códigos de longitud variable no deben verificarse juntos ya que, por ejemplo, 0, 01 y 001, etc. producen el mismo dígito de verificación. Sin embargo, todos los algoritmos de suma de comprobación son vulnerables a esto.

Algoritmo

La validez de una secuencia de dígitos que contiene un dígito de control se define sobre un quasigroup. Se puede tomar una tabla de cuasigrupos lista para usar de la disertación de Damm (páginas 98, 106, 111). Es útil si cada entrada diagonal principal es 0, porque simplifica el cálculo del dígito de control.

**Validar un número contra el dígito de control incluido**

- Configure un dígito interino e inicialícelo a 0.
- Procese el dígito número por dígito: use el dígito del número como índice de columna y el dígito intermedio como índice de fila, tome la entrada de la tabla y reemplace el dígito intermedio con este.
- El número es válido si y solo si el dígito intermedio resultante tiene el valor de 0.

**Cálculo del dígito de verificación**

**Requisito previo:** Las entradas diagonales principales de la tabla son 0.

- Configure un dígito interino e inicialícelo a 0.
- Procese el dígito número por dígito: use el dígito del número como índice de columna y el dígito intermedio como índice de fila, tome la entrada de la tabla y reemplace el dígito intermedio con este.
- El dígito interino resultante da el dígito de verificación y se agregará como dígito final al número.

Ejemplo:

Se utilizará la siguiente tabla de operación.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 1 | 7 | 5 | 9 | 8 | 6 | 4 | 2 |
| 1 | 7 | 0 | 9 | 2 | 1 | 5 | 4 | 8 | 6 | 3 |
| 2 | 4 | 2 | 0 | 6 | 8 | 7 | 1 | 3 | 5 | 9 |
| 3 | 1 | 7 | 5 | 0 | 9 | 8 | 3 | 4 | 2 | 6 |
| 4 | 6 | 1 | 2 | 3 | 0 | 4 | 5 | 9 | 7 | 8 |
| 5 | 3 | 6 | 7 | 4 | 2 | 0 | 9 | 5 | 8 | 1 |
| 6 | 5 | 8 | 6 | 9 | 7 | 2 | 0 | 1 | 3 | 4 |
| 7 | 8 | 9 | 4 | 5 | 3 | 6 | 2 | 0 | 1 | 7 |
| 8 | 9 | 4 | 3 | 8 | 6 | 1 | 7 | 2 | 0 | 5 |
| 9 | 2 | 5 | 8 | 1 | 4 | 3 | 6 | 7 | 9 | 0 |

Supongamos que elegimos el número (secuencia de dígitos) 572.

Cálculo del dígito de verificación

| digit to be processed → column index | 5 | 7 | 2 |
|---|---|---|---|
| old interim digit → row index | 0 | 9 | 7 |
| table entry → new interim digit | 9 | 7 | 4 |

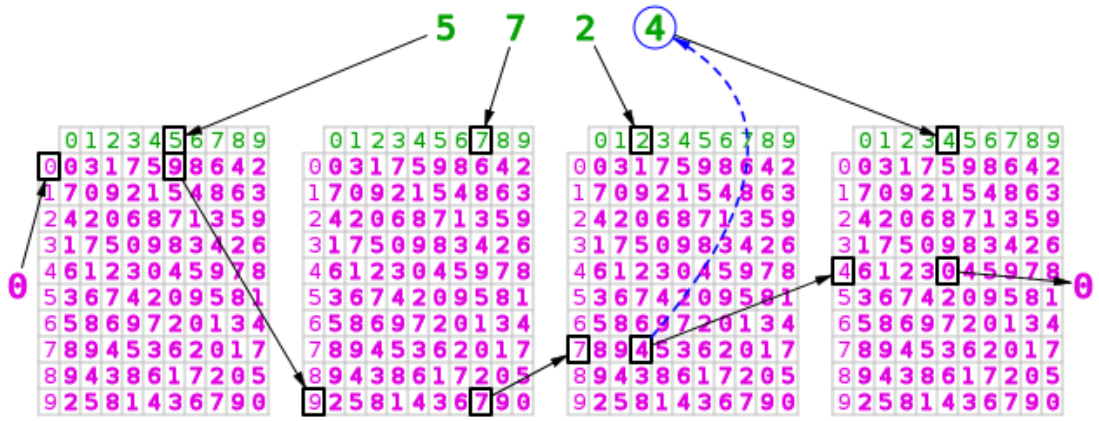
El dígito interino resultante es 4. Este es el dígito de verificación calculado. Lo agregamos al número y obtenemos 5724.

Validar un número contra el dígito de control incluido

| digit to be processed → column index | 5 | 7 | 2 | 4 |
|---|---|---|---|---|
| old interim digit → row index | 0 | 9 | 7 | 4 |
| table entry → new interim digit | 9 | 7 | 4 | 0 |

El dígito intermedio resultante es 0, por lo tanto, el número es válido.

Ilustración gráfica




### Algoritmo propio de Hash

Se implementó un algoritmo para que dado un set de números finito (la muestra puede contener números negativos o positivos), devuelve un hash en formato hexadecimal.

Ventajas de esta función hash:

- El sistema hexadecimal está en base 16, sus números están representados por los primeros 10 dígitos de la numeración decimal y el intervalo que va del número 10 al 15 están representados por las letras A a la F, lo que asegura que el hash sea más potente.
- La conversión de hexadecimal a binario es muy simple y no consume mucha memoria, ya que 16 = 2^4, cuatro números binarios componen un número hexadecimal.
- Incluso del sistema hexadecimal, podemos convertir a colores, lo que podría implementarse para hacer un hash "de colores".
- Antes del uso del número hexadecimal, se hacen operaciones con números primos. Los números primos solo pueden ser divididos exactamente por ellos mismos y por uno, lo que hace que se debe saber el número primo exacto si se quiere hacer una función inversa.

El algoritmo se encuentra en [https://github.com/abnerxch/Estructura-de-datos/tree/master/Hash%20functions/HashFunction%20-%20Abner](https://github.com/abnerxch/Estructura-de-datos/tree/master/Hash%20functions/HashFunction%20-%20Abner).

### Implementación de una Hash Table

El siguiente ejemplo utiliza una función hash simple, las colisiones se resuelven mediante sondeo lineal (estrategia de direccionamiento abierto) y la tabla hash tiene un tamaño constante. Este ejemplo muestra claramente los conceptos básicos de la técnica de hash.

La matriz subyacente tiene un tamaño constante para almacenar 20 elementos y cada ranura contiene un par clave-valor. La clave se almacena para distinguir entre pares clave-valor, que tienen el mismo hash.

Resolución de colisiones:

La *linear probing* se aplica para resolver colisiones. En caso de que la ranura, indicada por la función hash, ya haya sido ocupada, el algoritmo intenta encontrar una vacía al hacer *probing* en las ranuras consiguientes en la matriz.

La implementación se encuentra en https://github.com/abnerxch/Estructura-de-datos/tree/master/Hash%20functions/Hash%20table/src/main/java.

## Referencias


BSD Checksum. (2019). Retrieved from https://www.revolvy.com/page/BSD-checksum.

Blake (hash function) (2019). Retrieved fromhttps://en.wikipedia.org/wiki/BLAKE_(hash_function).

Damm Algorithm. (2019). Retrieved from https://en.wikipedia.org/wiki/Damm_algorithm.

Luhn algorithm. (2019). Retrieved from https://en.wikipedia.org/wiki/Luhn_algorithm.

Verhoeff algorithm. (2019). Retrieved from https://en.wikipedia.org/wiki/Verhoeff_algorithm.