Encriptación de mensajes

Otoño 2021

Contenidos:

1. Enunciado de la práctica

La Criptografía es la ciencia que trata sobre encriptar y desencriptar información. Desde la edad antigua se han encriptado mensajes diplomáticos, militares o administrativos para conseguir comunicaciones seguras, a salvo de espías.

En la actualidad, la Criptografía juega un papel de gran importancia dentro de la Informática. Su uso es prácticamente cotidiano, ya que elementos tan familiares como los passwords, los números de tarjetas, etc., se guardan encriptados e incluso hay programas que permiten cifrar y descifrar los mensajes de correo electronico y, en general, cualquier documento en soporte digital.

Un mensaje puede encriptarse por *permutación* o por *sustitución* de sus caracteres, siguiendo unas reglas predefinidas y quizá introduciendo algún elemento adicional (claves, números aleatorios, ...). Obviamente, si se desea descifrar un mensaje ya encriptado, resulta esencial que las reglas de cifrado sean reversibles.

En esta práctica proponemos dos métodos de encriptación, uno de cada tipo. Para cada caso solo mostramos el algoritmo de encriptación, el de desencriptación es fácilmente deducible.

1.1. Reordenación por rejillas

Una *rejilla* de dimensión n con k *huecos* (para abreviar, rejilla n_k) es un conjunto de k posiciones de una matriz $[1..n, 1..n]$. Cada posición (i, j) es lo que llamamos un *hueco*.

Ejemplo: el conjunto (1,3) (3,2) (2,4) (4,1) es una rejilla 4_4. Notad que en ningún momento se habla de los contenidos de la matriz, sólo de posiciones.

Decimos que una rejilla n_k es *válida* para nuestro método de encriptación si $k = n^2/4$ y al obtener las rejillas resultantes de girar la original 90, 180 y 270 grados en el mismo sentido, no se repite ningún hueco. Dicho de otro modo, si los $4k$ huecos de la unión de las cuatro rejillas cubren las n^2 posiciones de la matriz.

Dada una rejilla n_k válida, el texto a encriptar se divide en bloques de tamaño n^2. Si el último bloque no se llena, se rellena con algún carácter especial. Cada bloque se encripta por separado empleando la rejilla.

Un bloque de n^2 caracteres se encripta con una rejilla n_k válida, de la siguiente manera. Se leen los k primeros caracteres del bloque y se copian en una matriz vacía, ocupando las posiciones marcadas por los huecos de la rejilla. Los huecos se ocupan de izquierda a derecha y de arriba a abajo respecto a su posición en la matriz (visualmente: $(1, 1)$ = extremo superior izquierdo; (n, n) = extremo inferior derecho). A continuación, se gira la rejilla (no la matriz) 90 grados en contra de las agujas de reloj, con lo que aparecen k huecos nuevos. Dichos huecos se ocupan con los siguientes caracteres, con cuidado de hacerlo de izquierda a derecha y de arriba a abajo (al girar, los huecos pueden no conservar el orden). Después se repite el proceso con el segundo giro y por último con el tercero. Al terminar las cuatro fases, todo el bloque original estará copiado en la matriz. El bloque encriptado es el resultado de recorrer dicha matriz fila por fila en el orden habitual. Obviamente, queda una permutación del bloque original.

Por ejemplo, con la rejilla anterior el bloque formado por los 16 primeros caracteres del texto "`Hola, amigo, qué tal?`" sufre la transformación mostrada en la siguiente figura. Notad que hemos usado un símbolo especial para el carácter blanco, de modo

que no se confunda con una casilla no ocupada, y que hemos destacado los huecos producidos en cada fase.

Bloque original

H	o	l	a
,	␣	a	m
i	g	o	,
␣	q	u	e

Las cuatro fases de la ocupación de huecos

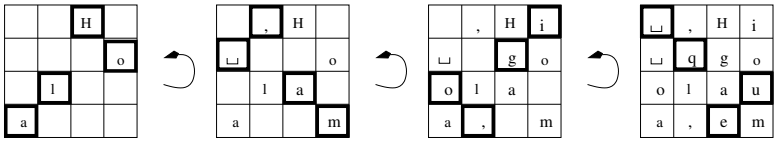

Figura 1

El texto encriptado queda `" ,Hi qgoolaua,ém"`.

Si aplicamos el proceso sobre cada bloque, el texto original se convertirá en un conjunto de $\lceil long\_texto/n^2 \rceil$ bloques encriptados. Repetimos que el último bloque quizá no se llene con caracteres del texto, en cuyo caso lo rellenaremos convenientemente antes de encriptarlo. Si el texto es vacío no se encripta ningún bloque.

1.2. Sustitución por patrones arborescentes

Un *patrón* es un árbol binario formado por enteros no negativos, pertenecientes a un cierto intervalo, que depende del rango de caracteres permitidos en los textos a encriptar.

Un mensaje se puede encriptar mediante un patrón de la siguiente manera. En primer lugar (paso 1) el mensaje se transforma en un árbol binario de caracteres lo más completo posible, es decir, llenando cada nivel antes de pasar al siguiente. Cada nivel se llena ocupando sus nodos de izquierda a derecha según se recorre el mensaje.

Ejemplo: consideremos el texto `"Cuando duerma con la soledad."`. La siguiente figura muestra el árbol resultante y un posible patrón.

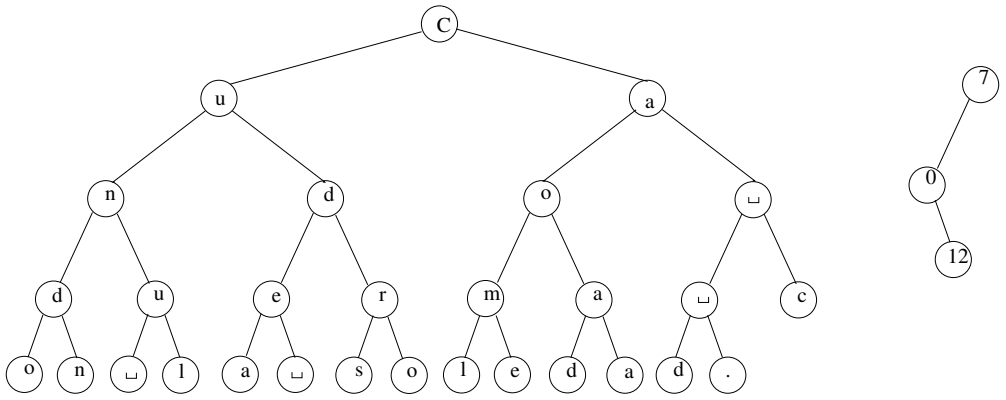

Figura 2: Un árbol-mensaje y un patrón

Una vez copiado el mensaje en el árbol, se obtiene un segundo árbol, esta vez de enteros, con la misma forma que el primero, construido a base de replicar el patrón total o parcialmente cuantas veces sea necesario y usando trozos lo más grandes posible comenzando por la raíz (paso 2). Al segundo árbol le llamamos *mosaico*. En la siguiente figura mostramos el mosaico obtenido a partir del mensaje anterior. Notad, señaladas en rojo, las apariciones del patron completo y en azul las del patrón parcial de dos elementos. El resto son apariciones del patrón parcial de un elemento, es decir, la raíz.

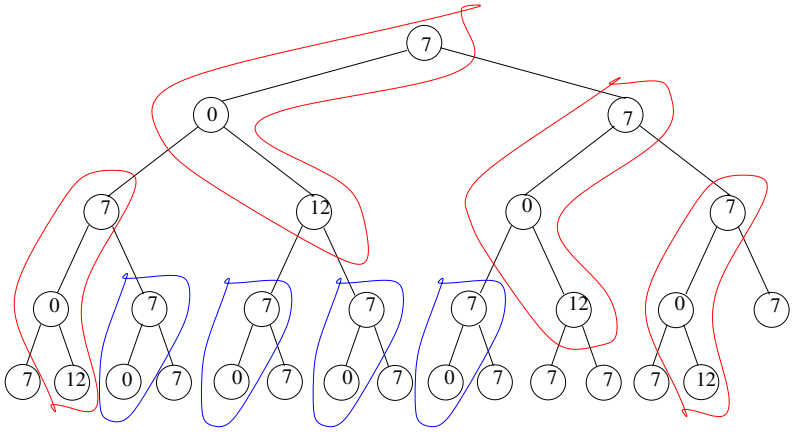

Figura 3: El mosaico resultante

Por último (paso 3), sustituimos cada carácter del árbol-mensaje por el resultado de sumarle circularmente el entero situado en su posición correspondiente en el mosaico. El texto encriptado se obtiene deshaciendo la transformación del paso 1 a partir de este tercer árbol.

En el caso del ejemplo, el resultado es `"Juhupo'd|lytm jvz sa'vllkhk:"`.

La encriptación mediante patrones arborescentes también se puede aplicar a bloques consecutivos del mensaje, similarmente al caso de encriptación por rejilla, dado un cierto valor b (el tamaño del bloque):

- Si $b \geq long\_texto$, se aplica el método que acabamos de describir, sin necesidad de rellenar el bloque con caracteres especiales, es decir, el árbol-mensaje seguiría teniendo un tamaño $long\_texto$.
- Si $b < long\_texto$, entonces el mensaje se divide en $\lceil long\_texto/b \rceil$ bloques, todos de tamaño b excepto el último, que puede tener un tamaño menor o igual a b. Entonces, cada bloque se encripta por separado empleando el mismo mosaico de tamaño b obtenido a partir del patrón.

2. Funcionalidades

2.1. Decisiones sobre los datos

Con el objetivo de simplificar los procesos de encriptación y desencriptación, los mensajes que usaremos para probar la práctica estarán formados únicamente por carac-

teres de la tabla ASCII con índices entre 32 (espacio en blanco) y 126 (∼). En particular, eso implica que

- un mensaje estará formado por una sola línea de texto
- en la encriptación mediante patrones arborescentes solo emplearemos patrones formados por enteros entre 0 y 94
- la suma circular que nos interesa para codificar un caracter c a partir de un entero d que aparece en un patrón será $32 + (c + d - 32) \ \% \ 95$

Los datos del programa se dividen en dos partes. Un primer grupo de mensajes, rejillas y patrones se usará para la inicialización del sistema. Una vez realizada dicha inicialización, el programa ofrecerá un repertorio de funcionalidades. Cada funcionalidad podrá requerir a su vez unos datos y producirá una salida que describimos con más precisión en el siguiente apartado.

En todo momento, se leerán solo datos sintácticamente correctos y no será necesario comprobar dicha corrección. Por ejemplo, si decimos que inicialmente el sistema contiene M mensajes, el programa recibirá un entero M mayor o igual que 0 y a continuación exactamente M mensajes, con las restricciones mencionadas a lo largo de este enunciado. Las rejillas y los patrones serán asimismo sintácticamente correctos. En el caso de las rejillas, si leemos la dimensión y la longitud de una rejilla n_k, entonces n y k serán enteros mayores que 1 y 0 respectivamente, a continuación se leerán k posiciones y dichas posiciones serán pares de enteros entre 1 y n. Además, se garantiza que las rejillas que se leen en la inicialización serán válidas de cara al método de encriptación que vamos a utilizar.

2.2. Programa principal: estructura y comandos

El programa principal empezará leyendo un número inicial de mensajes $M \geq 0$ y una secuencia de M mensajes. Cada mensaje vendrá precedido por un string, que servirá como identificador del mismo.

A continuación se leerá un número inicial de rejillas $R \geq 0$ y una secuencia de R rejillas, que se guardarán en orden de lectura con identificadores $1 \ldots R$. Para cada rejilla se leerá su dimensión n, su tamaño k y sus k posiciones.

Por último se leerá un número inicial de patrones $P \geq 0$ y una secuencia de P patrones, que se guardarán en orden de lectura con identificadores $1 \ldots P$. Para cada patrón se leerán sus valores en preorden con marca -1.

Hay que notar que las magnitudes M, R y P no serán constantes ya que podrán aumentar al añadir posteriormente nuevos mensajes, nuevas rejillas o nuevos patrones. Nunca se borrarán ni rejillas ni patrones, y una vez añadidos, no se podrán modificar los mensajes, las rejillas o los patrones.

Terminadas las inicializaciones, se procesará una serie de comandos acabados en un comando `fin`. La estructura general del programa principal será la siguiente:

```
lectura del conjunto inicial de mensajes;
lectura del conjunto inicial de rejillas;
```

```
lectura del conjunto inicial de patrones;
lee comando;
while (comando != "fin") {
    procesa comando;
    lee comando;
}
```

Los comandos aceptados se describen a continuación. Todo ellos se presentan en dos versiones, una con el nombre completo y otra con el nombre abreviado. La sintaxis exacta de la entrada y la salida de cada comando se podrá derivar del juego de pruebas público del ejercicio creado en el Jutge para cada entrega.

1. `nuevo_mensaje` idm: lee y añade un nuevo mensaje con identificador idm. El comando admite la forma abreviada `nm`. Si ya existía un mensaje en el sistema con el mismo identificador se imprime un mensaje de error. En caso contrario se imprime el identificador del mensaje y el número de mensajes M en el sistema después de añadirlo.

2. `nueva_rejilla`: lee una nueva rejilla y, si es válida para encriptar mensajes, la añade al sistema, tomando como identificador el número de rejillas previamente existentes más uno. El comando admite la forma abreviada `nr`. Primero se debe leer la dimensión y la longitud de la rejilla, de la misma manera que se hizo en las rejillas iniciales, y a continuación sus posiciones. Si la dimensión y la longitud no son consistentes se imprime un mensaje de error. Si lo son, pero aún así la rejilla no es válida para encriptar mensajes, se imprime otro mensaje de error. En caso contrario se imprime el número de rejillas R en el sistema después de añadirla.

3. `nuevo_patron`: lee y añade un nuevo patrón, que tendrá como identificador el número de patrones previamente existentes más uno. El comando admite la forma abreviada `np`. Además, se imprime el número de patrones P en el sistema después de añadirlo.

4. `borra_mensaje` idm: elimina del sistema un mensaje con identificador idm. El comando admite la forma abreviada `bm`. Si no existe un mensaje con identificador idm se imprime un mensaje de error. En caso contrario se imprime el número de mensajes M después de borrarlo. Si más tarde se añade otro mensaje con el mismo nombre es como si el anterior no hubiera existido.

5. `listar_mensajes`: se listan los mensajes del sistema en orden alfabético de identificador. El comando admite la forma abreviada `lm`.

6. `listar_rejillas`: se listan las rejillas del sistema en orden creciente de identificador. El comando admite la forma abreviada `lr`. Cada rejilla ha de ir acompañada de sus huecos originales y los producidos en los tres giros.

7. `listar_patrones`: se listan los patrones del sistema en orden creciente de identificador. El comando admite la forma abreviada `lp`.

8. `codificar_rejilla` idr: lee y codifica un mensaje mediante la rejilla con identificador idr. El comando admite la forma abreviada `cr`. Si la rejilla idr no existe se imprime un mensaje de error. En caso contrario se imprime el mensaje codificado. Si el último bloque del mensaje no se llena, se completa con caracteres blancos (' ') antes de codificarlo.

9. `codificar_guardado_rejilla` idm idr: codifica el mensaje con identificador idm mediante la rejilla con identificador idr. El comando admite la forma abreviada `cgr`. Si el mensaje idm no existe se imprime un mensaje de error. Si la rejilla idr no existe se imprime un mensaje de error. En caso contrario se imprime el mensaje codificado. Si el último bloque del mensaje no se llena, se completa con caracteres blancos (' ') antes de codificarlo.

10. `decodificar_rejilla` idr: lee y decodifica un mensaje mediante la rejilla con identificador idr. El comando admite la forma abreviada `dr`. Si la rejilla idr no existe se imprime un mensaje de error. Si existe, pero la longitud del mensaje a decodificar no es un múltiplo del cuadrado de la dimensión de la rejilla, se imprime otro mensaje de error. En caso contrario se imprime el mensaje decodificado. Este tendrá siempre la misma longitud que el mensaje codificado, que no ha de coincidir necesariamente con la del original, ya que durante la codificación se han podido añadir caracteres blancos al final del mensaje, que han de conservarse al decodificar.

11. `codificar_patron` idp b: lee y codifica un mensaje mediante el patrón con identificador idp, dividiendo el mensaje en bloques de tamaño b. El comando admite la forma abreviada `cp`. Si el patrón idp no existe se imprime un mensaje de error. En caso contrario se imprime el mensaje codificado.

12. `codificar_guardado_patron` idm idp b: codifica el mensaje con identificador idm mediante el patrón con identificador idp, dividiendo el mensaje en bloques de tamaño b. El comando admite la forma abreviada `cgp`. Si el mensaje idm no existe se imprime un mensaje de error. Si el patrón idp no existe se imprime un mensaje de error. En caso contrario se imprime el mensaje codificado.

13. `decodificar_patron` idp b: lee y decodifica un mensaje mediante el patrón con identificador idp, dividiendo el mensaje en bloques de tamaño b. El comando admite la forma abreviada `dp`. Si el patrón idp no existe se imprime un mensaje de error. En caso contrario se imprime el mensaje decodificado.

14. `fin`: termina la ejecución del programa