TECNOLOGÍA DE TRANSMISIÓN INALÁMBRICA POR MEDIO DE ONDAS SONORAS

PROYECTO FINAL

NELSON QUIÑONES - CARLOS H. GONZALEZ - PAOLA VELOZA - JAIME CARDONA
COMUNICACIONES DIGITALES
UNIVERSIDAD ICESI

OBJETIVOS

GENERAL:

El objetivo general es el de desarrollar y caracterizar el desempeño de una aplicación de comunicación inalámbrica que use el sonido para transmitir datos. Este prototipo se realiza como prueba previa al desarrollo de la aplicación móvil. Por lo tanto, puede realizarse en un computador usando un micrófono y parlantes. Lo que se desea lograr es caracterizar el sistema y probarlo en ambientes ruidosos para así verificar su viabilidad como solución tecnológica.

REQUERIMIENTOS FUNCIONALES:

1. Los datos deben transmitirse con cada boleta digital, no solo está asociada a un número único de la boleta, sino también a datos personales del asistente para verificar su identidad y permitir el ingreso.

2. Los datos que se deben transmitir cuando va a ingresar un asistente son:
   a. Número único de la boleta (10 dígitos)
   b. Nombres y Apellidos del asistente.
   c. Documento de identidad.
   d. Edad y Género.
   e. Fotografía reciente (menos de 6 meses) (180x180 píxeles) de frente.

3. Los datos se deben transmitir por ondas sonoras desde un smartphone (TX) con aplicación móvil asociada al asistente y dispositivo receptor para el personal de seguridad permitir el ingreso.

TABLA DE CONTENIDO

c. INFORMACIÓN DE LA CODIFICACIÓN DE FUENTE:
   i. 1) Calcular y mostrar cuántos bits tiene la información original de la imagen.
   ii. 2) Calcular y mostrar cuántos bits tiene la información codificada de la imagen.
   iii. 3) Calcular y mostrar cual es la relación de compresión.

CONTENIDO

1. MANEJO DE ARCHIVOS EN EL PC:
   a. Desarrollar un código que pueda leer un texto plano (en formato ASCII) desde un archivo del computador.

```
43          /**
44           * Metodo para leer información de un archivo de texto plano y almacenar el contenido en <b>cadena</b>
45           * @throws IOException
46           * @throws FileNotFoundException
47           */
48          public void lecturaArchivo() throws IOException, FileNotFoundException {
49
50                  FileReader file = new FileReader(direccion);
51                  BufferedReader br = new BufferedReader(file);
52                  String temp = "";
53                  StringBuilder constructor = new StringBuilder();
54                  while((temp = br.readLine())!= null){
55                          constructor.append(temp);
56                  }
57                  cadena = constructor.toString();
58                  br.close();
59          }
```

Figura 1. Lectura de archivo

El anterior método se encarga de leer un texto plano en formato ASCII desde un archivo de computadora y almacena el contenido en una cadena de caracteres llamada cadena.

   b. Desarrollar un código que pueda leer una imagen (en formato .jpg) desde un archivo del computador.

```java
    /**
 * Constructor de Imagen
 * @param imagePath : String es la ruta de la imagen con que trabajara la instancia de {@link #Imagen}
 */
    public Imagen(String imagePath) {
            this.imagePath = imagePath;
            image = IJ.openImage(imagePath);
            calculatePixInformation();
            //image.show();
    }

     /**
 * Constructor de Imagen
 *
 * @param imageInfo : String[] es un arreglo con la información de cada pixel de la imagen
 * con el que se creara una instancia de {@link #Imagen}.
 *
 * imageInfo[] se puede obtener del String <b>bitInformation</b>
 * separando su contenido por comas.
 */
    public Imagen(String imageInfo[]) {
            int altura = Integer.parseInt(imageInfo[0]);
            int ancho = Integer.parseInt(imageInfo[1]);

            ImagePlus ref = new ImagePlus();
            ImageProcessor myProcessor = ref.getProcessor();


            for(int i = 0; i<altura ; i++) {
                    for(int j = 0; j<ancho; j++) {
                            myProcessor.set(i, j, Integer.parseInt(imageInfo[i*ancho+j]));
                    }
            }

            this.image = ref;
            //image.show();
    }
```

Figura 2. Constructores de la clase imagen.

Evidenciamos los constructor de la clase imagen, los cuales se encargan de leer imágenes en formato jpg desde un archivo existente en el computador.
Existen 2 maneras de hacerlo:

- Por ruta.
- Por un arreglo de pixeles.

c. Desarrollar un código que pueda leer información binaria desde un archivo del computador.

```java
    /**
     * @throws FileNotFoundException
     * @throws IOException
     */
    public void lecturaArchivoTextoBinario() throws FileNotFoundException, IOException {
            lecturaArchivo();
            cadena = traducirBinario(cadena);
    }
```

Figura 3. Lectura de archivo de texto binario.

Para el desarrollo del proyecto creamos un método capaz de leer archivos en formato binario desde un archivo de computador sin ningún problema, a este método le evaluamos:

- excepción de que el archivo si se encuentre.
- excepción de entrada y salida.

```
106         /**
107          * Metodo para  traducir un texto binario a un texto alfanumérico
108          * @param s : String con el texto a traducir
109          * @return : Un String con el texto alfanumérico
110          */
111         private String traducirBinario(String s) {
112                 StringBuilder out = new StringBuilder();
113         String chars[] = s.split(" ");
114
115         for(int i = 0; i<chars.length; i++) {
116                 out.append((char)Long.parseLong(chars[i],2));
117         }
118
119                 return out.toString();
120         }
```

Figura 4 . Traducir Binario.

Este método es encargado de traducir un texto binario a un texto alfanumérico. Tiene como parámetro una cadena de caracteres la cual es el texto en formato binario inicialmente y luego se procesa para su formato final.

d. Desarrollar un código que guarde información en binario en un archivo del computador.

```
97          /**
98           * Metodo para crear un archivo con la información de <b>info</b> traducida a binario
99           * @param nombre : Un String con el nombre que se le dara al archivo
100          * @param info  : Un String con la informacion que se quiere poner el archivo
101          * @throws IOException
102          */
103         public void crearArchivoInfoBinaria(String nombre,String info) throws IOException {
104                 String contenido = generarBinario(info);
105                 crearArchivo(nombre,contenido);
106         }
107
```

Figura 5. Creación de archivo con información traducida a binaria.

Mediante este método se evidencia el proceso de guardar información binaria en un archivo del computador, este archivo tiene un nombre y la información son los datos a guardar en binario.

2. PARA EL TEXTO:
    a. CODIFICACIÓN:
        i. Desarrollar un código que transforme el texto plano (en formato ASCII) a información binaria usando el algoritmo LZW visto en clase.

```java
    /**
     * Metodo para codificar el String <b>mensajeOriginal</b> utilizando el algoritmo LZW,
     * el resultado se almacena en el String <b>mensajeCodificado</b> con la siguiente estructura:
     * <p><b>tabla LZW + "*" + mensaje codificado</b>.</p>
     * El mensaje codificado va a ser un String con una serie de enteros separados por comas
     * que representan la salida del algoritmos LZW.
     */
    public void codificarLZW() {
        HashMap<String,Integer> lookUp = new HashMap();
        String info[] = mensajeOriginal.split("");      //abc a,b,c
        StringBuilder out = new StringBuilder();        //Es más rapido

        out.append(creacionTablaLZW(lookUp,info));
        int id =lookUp.size();

        out.append("*");

        String p = "";

        for(int i = 0; i<info.length; i++) {
            String c = info[i];
            if(lookUp.containsKey(p+c)) {
                p = p+c;
            }else {
                out.append(lookUp.get(p));out.append(",");
                lookUp.put(p+c,id);id++;
                p = c;
            }
        }
        out.append(lookUp.get(p));
        mensajeCodificado = out.toString();
        bitsMensajeCodificado = mensajeCodificado.length()*7;
    }
```

Figura 6. Codificación LZW

```java
/**
 * El metodo crea un String que representa la tabla LZW que se va a utilizar para codificar.<p></p>
 * El String esta compuesto de parejas de un caracter unico del mensaje separado por una coma del entero que lo representa <b>ej : a
 * a su vez cada pareja va a estar separada entre si por un punto y coma <b>ej : a,0;b,1;c,2</b>.
 *
 * @param lookUp : El HashMap en el que se representara la tabla LZW para ejecutar el algoritmo.
 * @param info : Un arreglo de Strings con cada caracter del mensaje a codificar.
 *
 * @return un String con la con la tabla LZW.
 */
public String creacionTablaLZW(HashMap<String,Integer> lookUp,String info[]) {
    int id = 0;
    StringBuilder out = new StringBuilder();
    for(int i = 0; i<mensajeOriginal.length(); i++) {
        if(!lookUp.containsKey(info[i])) {
            lookUp.put(info[i],id);
            out.append(info[i]);
            out.append(",");
            out.append(id);
            out.append(";");
            id++;
        }
    }
    return out.toString();
}
```

Figura 7. Creación tabla LZW

Para el desarrollo del código (Figura 6) se utiliza un arreglo de String (linea 58) el cual va a contener los caracteres del texto plano en formato ASCII. Además, se utiliza una estructura de java llamada HashMap (línea 57), la cual permite organizar un valor y una clave, en donde apoyándonos en el método creacionTablaLZW (Figura 7) el valor va a corresponder a un carácter (posición) del arreglo de String y la clave a un identificador de dicho carácter. De esta forma, de la línea 96 a la línea 104 se recorre el arreglo (MensajeOriginal) para llenar los valores del HashMap. Seguidamente, apoyandonos del pseudocódigo presentado en clase del algoritmo LZW se realiza la codificación de la linea 67 a la 79. Finalmente la línea 80 indica el String con el mensaje codificado.

ii. Mostrar el resultado binario en pantalla.

Para evidenciar si los algoritmos funcionan lo hicimos a través de pruebas unitarias.

```
@Test
public void testLZWAlgorithm2() {
    System.out.println("\n\n/////////////////////////////////////// TEST LZW 2 ///////////////////////////////////////");
    Scene5();
    codificador = new Codificacion(mensaje2,0);
    codificador.codificarLZW();
    System.out.println("Mensaje codificado LZW: "+codificador.getMensajeCodificado());
    mensaje = codificador.getMensajeCodificado();
    codificador2 = new Codificacion(mensaje, 1);
    codificador2.decodificarLZW();
    System.out.println("Mensaje decodificado LZW: "+codificador2.getMensajeOriginal());

}
```

Figura 8. Test codificador LZW.

```
/////////////////////////////////////// TEST LZW 2 ///////////////////////////////////////
Mensaje codificado LZW: t,0;r,1;e,2;s,3; ,4;i,5;g,6;c,7;o,8;m,9;a,10;n,11;*0,1,2,3,4,12,5,3,0,14,16,5,6,13,15,7,8,9,5,10,11,16,1,23,8
Mensaje decodificado LZW: tres tristes tigres comian trigo
```

Figura 9. Consola de la prueba.

En la figura 9 podemos observar como trabaja el método de codificación de LZW, vemos que finalmente si se crea la tabla con los caracteres de la frase, después de mostrarnos la tabla, nos muestra la frase codificada en LZW.

b. DECODIFICACIÓN DE TEXTO:
   i. Desarrolle un código que transforme la información binaria a texto plano (en formato ASCII) usando el algoritmo LZW visto en clase.

```
121  * Metodo para decodificar el String <b>mensajeCodificado</b> utilizando el algoritmo LZW,
122  * el resultado se almacena en el String <b>mensajeOriginal</b>.
123  */
124 public void decodificarLZW() {
125     String rawInfo[] = mensajeCodificado.split("\\*");
126     String rawTable[] = rawInfo[0].split(";");
127     String rawMessage[] = rawInfo[1].split(",");
128 
129     HashMap<Integer,String> lookUp = new HashMap();
130 
131     int id = 0;
132 
133     for(int i = 0; i<rawTable.length;i++) {
134         String symbol = rawTable[i].split(",")[0];
135         id = Integer.parseInt(rawTable[i].split(",")[1]);
136         lookUp.put(id,symbol);
137     }
138 
139     StringBuilder out = new StringBuilder();
140 
141     int o = Integer.parseInt(rawMessage[0]);
142     String s = "";
143     String c = "";
144     out.append(lookUp.get(o));
145 
146     for(int i = 1; i<rawMessage.length;i++) {
147         int n = Integer.parseInt(rawMessage[i]);
148 
149         if(!lookUp.containsKey(n)){
150             s = lookUp.get(o);
151             s += c;
152         }else {
153             s = lookUp.get(n);
154         }
```

Figura 10. Decodificación LZW parte 1

```
141     int o = Integer.parseInt(rawMessage[0]);
142     String s = "";
143     String c = "";
144     out.append(lookUp.get(o));
145 
146     for(int i = 1; i<rawMessage.length;i++) {
147         int n = Integer.parseInt(rawMessage[i]);
148 
149         if(!lookUp.containsKey(n)){
150             s = lookUp.get(o);
151             s += c;
152         }else {
153             s = lookUp.get(n);
154         }
155 
156         out.append(s);
157         c = s.charAt(0)+"";
158         id++;
159         lookUp.put(id,lookUp.get(o)+c);
160         o = n;
161     }
162 
163     mensajeOriginal = out.toString();
164     bitsMensajeOriginal = mensajeOriginal.length()*7;
165 }
```

Figura 10. Decodificación LZW parte 2.

Para el desarrollo del código de la Figura , utilizamos tres arreglos de String, el primer arreglo nos separa la tabla LZW y la frase codificada de LZW, el segundo arreglo guarda la tabla LZW, y el último arreglo guarda la codificación de LZW, después con la

ayuda del segundo arreglo que contiene la tabla LZW (rawTable) crea un HashMap que nos ayudará a tener mejor manejo de esta tabla LZW, después se realiza el proceso ya conocido para la decodificación LZW, es decir vamos viendo la frase codificada, agregamos nuevos elementos si no están en la tabla y finalmente obtenemos la frase.

ii. Mostrar el resultado en pantalla.

```
public void testLZWAlgorithm1() {
    System.out.println("\n\n////////////////////////////////////// TEST LZW 1 //////////////////////////////////////");
    Scene2();
    codificador = new Codificacion(mensaje,0);
    codificador.codificarLZW();
    System.out.println("Mensaje codificado LZW: "+codificador.getMensajeCodificado());
    mensaje = codificador.getMensajeCodificado();
    codificador2 = new Codificacion(mensaje, 1);
    codificador2.decodificarLZW();
    System.out.println("Mensaje decodificado LZW: "+codificador2.getMensajeOriginal());

}
```

Figura 11. Prueba de codificación LZW.

```
////////////////////////////////////// TEST LZW 1 //////////////////////////////////////
Mensaje Original: como coco
Mensaje codificado LZW: c,0;o,1;m,2; ,3;*0,1,2,1,3,4,4
Mensaje decodificado LZW: como coco
```

F*igura 12. Impresión en pantalla de decodificación LZW.*

Como podemos observar el decodificado de LZW no estrega la frase exacta a la que ingresamos para hacer la prueba, es decir nuestro algoritmo LZW funciona correctamente

c. INFORMACIÓN DE LA CODIFICACIÓN DE FUENTE:

i. Calcular y mostrar cuántos bits tiene la información original del texto.

```
25       */
26      private int bitsMensajeOriginal;
27      /**
28       * Un entero con la cantidad de bits de la información codificada
29       */
30      private int bitsMensajeCodificado;
31
32      /**
```

Figura 13. Atributos cálculo de bits original y codificado

```
266     public int getBitsMensajeOriginal() {
267         return bitsMensajeOriginal;
268     }
```

Figura 14. Método para obtener los bits originales

Para conocer cuántos bits tiene la información original, se creó un atributo en la clase Codificación llamado bitsMensajeOriginal, en el cual se van a almacenar dichos bits.

Para el cálculo de este atributo, no se desarrolló un método como tal, si no que este se va calculando a medida que se va codificando en cada algoritmo, como se puede evidenciar en la línea 91, de la figura 6.

ii. Calcular y mostrar cuántos bits tiene la información codificada del texto.

Teniendo en cuenta la figura , para el cálculo de los bits que tiene la información codificada del texto, se creó un atributo (línea 30) que permite almacenar los bits. Esto, al igual que el punto anterior no se desarrolla dentro de un método como tal, si no que se va calculando a medida que se codifica o se decodifica. Al final, para acceder a este atributo se hace a través de un método get.

```
270    public int getBitsMensajeCodificado() {
271        return bitsMensajeCodificado;
272    }
273 }
```

Figura 15. Método para obtener el número de bits codificados.

iii. Calcular y mostrar cual es la relación de compresión.

```
231 /**
232  * El metodo calcula la relación de compresión entre <b>bitsMensajeOriginal</b> y <b>bitsMensajeCodificado<
233  * @return un numero real representando la relasión de compresion
234  */
235 public double calcularRelacionCompresion() {
236     return (double)bitsMensajeOriginal/(double)bitsMensajeCodificado;
237 }
238
```

Figura 16. Cálculo de RC.

Para el cálculo de la relación de compresión se utilizan los atributos generados en la figura , estos dos atributos se dividen entre sí, y se obtiene a través de este método que devuelve un valor tipo double.

3. PARA LA IMAGEN:
    a. CODIFICACIÓN:
        i. Desarrollar un código que transforme la información de cada canal de color RGB a información binaria usando el algoritmo RLE visto en clase.

```java
    /**
     * Metodo para aplicar el algoritmo de codificacion RLE sobre el String
     * de <b>mensajeOriginal</b> y almacenar el resultado en el String <b>mensajeCodificado</b>
     */
    public void codificarRLE() {
        String info = mensajeOriginal;
        StringBuilder result = new StringBuilder();
        int pos = 0;
        int accum = 1;

        while(pos < info.length()-1) {
            if(info.charAt(pos) == info.charAt(pos+1)) {
                accum++;
            }else {
                result.append(accum);
                result.append(info.charAt(pos));
                result.append('-');
                accum = 1;
            }
        }

        result.append(accum);
        result.append(info.charAt(info.length()-1));

        mensajeCodificado =  result.toString();
    }
```

Figura 17. Codificación RLE.

Para la codificación del código de la Figura 17, se utilizó un String que contenga la información original y un StringBuilder, debido a que este permite construir cadenas de texto de una forma más eficiente. Y seguido a esto con base en el pseudocódigo visto en clase, se realiza la codificación del algoritmo RLE.

ii. Mostrar el resultado en pantalla.

```java
@Test
public void testRLEAlgorithm1() {
    Scene1();
    codificador = new Codificacion(im.getBinInformation(), 0);
    codificador.codificarRLE();
    codificador2 = new Codificacion(codificador.getMensajeCodificado(),1);
    codificador2.decodificarRLE();
    System.out.println("\n\n/////////////////////////////////////////// TEST RLE 1 ///////////////////////////////////////////");
    System.out.println("Mensaje codificado RLE: "+codificador.getMensajeCodificado());
    System.out.println("Mensaje decodificado RLE: "+codificador2.getMensajeOriginal());
    assertEquals(codificador2.getMensajeOriginal(),im.getBinInformation());

}
```

Figura 18. Prueba codificación RLE.

```
Mensaje codificado RLE: 1,423;1,648;4,-1;2,-263173;13,-1;1,-263173;600,-1;1,-263173;13,-1;2,-263173;16,-1;2,-263173;13,-1;1,-263173;600,-1
```

Figura 19. Impresión en pantalla de codificación RLE.

Vemos cómo el mensaje codificado cuenta las veces que se repite un dato consecutivamente, y lo almacena de la siguiente forma 1 (el número de veces que se repite), 423 (el dato). Para poder observar esto mejor realizamos otro test con texto para ver mejor el funcionamiento del algoritmo.

```java
@Test
public void testRLEAlgorith2() {
    Scene3();
    codificador = new Codificacion(mensaje, 0);
    codificador.codificarRLE();
    System.out.println("\n\n////////////////////////////////////// TEST RLE 2 ////////////////////////,
    System.out.println("Mensaje codificado RLE: "+codificador.getMensajeCodificado());
    mensaje = codificador.getMensajeCodificado();
    codificador2 = new Codificacion(mensaje, 1);
    codificador2.decodificarRLE();
    System.out.println("Mensaje decodificado RLE:"+codificador2.getMensajeOriginal());
}
```

Figura 20. Prueba 2.

```
Mensaje Original: a,a,a,a,a,a,e, ,l,a,s, ,r,e,p,e,t,i,c,i,o,n,e,s, ,e,n, ,m,i,s, ,t,r,a,d,u,c,c,i,o,n,e,s

////////////////////////////////////// TEST RLE 2 //////////////////////////////////////
Mensaje codificado RLE: 6,a;1,e;1, ;1,l;1,a;1,s;1, ;1,r;1,e;1,p;1,e;1,t;1,i;1,c;1,i;1,o;1,n;1,e;1,s;1, ;1,e;1,n;1, ;1,m;1,i;1,s;1, ;1,t;1,r;1,
Mensaje decodificado RLE:a,a,a,a,a,a,e, ,l,a,s, ,r,e,p,e,t,i,c,i,o,n,e,s, ,e,n, ,m,i,s, ,t,r,a,d,u,c,c,i,o,n,e,s,
```

Figura 21. Impresión en pantalla de prueba 2.

Aquí observamos que la a ser repite 6 veces consecutivamente, el algoritmo las acumula correctamente, en lo que queda de la secuencia no hay repeticiones consecutivas, es decir todas en adelante serán 1 como se muestra en la consola, es decir el algoritmo de codificación del RLE funciona correctamente.

b. DECODIFICACIÓN DE IMAGEN:
    i. Desarrollar un código que transforme la información binaria a información de color RGB para cada canal usando el algoritmo RLE visto en clase.

```java
191    /**
192     * Metodo para decodificar el String <b>mensajeCodificado</b> utilizando el algoritmo RLE,
193     * el resultado se almacena en el String <b>mensajeOriginal</b>,
194     * separando cada elemento del mensaje original por una coma.
195     */
196    public void decodificarRLE() {
197        String info = mensajeCodificado;
198        StringBuilder result = new StringBuilder();
199        String splitInfo[] = info.split(";");
200
201        for (int i = 0; i < splitInfo.length; i++) {
202            String ref[] = splitInfo[i].split(",");
203            if (ref.length > 0) {
204                int repetitions = Integer.parseInt(ref[0]);
205                String symbol = ref[1];
206                for (int j = 0; j < repetitions; j++) {
207                    result.append(symbol);
208                    result.append(",");
209                }
210            }
211        }
212
213        mensajeOriginal = result.toString();
214    }
215
```

Figura 22. Decodificación RLE.

Para realizar el código de la Figura, se utiliza un String que contenga el mensaje Original (Línea 197) un StringBuilder en donde se construya la cadena decodificada (Línea 199) y un arreglo de String que contenga el mensaje caracter por caracter en

cada posición del arreglo. Seguido a esto, apoyados con el pseudocódigo visto en clase, se realizó la codificación del decodificador RLE.

ii. Mostrar el resultado en pantalla.

```
@Test
public void testRLEAlgorithm1() {
    Scene1();
    codificador = new Codificacion(im.getBinInformation(), 0);
    codificador.codificarRLE();
    codificador2 = new Codificacion(codificador.getMensajeCodificado(),1);
    codificador2.decodificarRLE();
    System.out.println("\n\n////////////////////////////////////////// TEST RLE 1 ////////////////////////");
    System.out.println("Mensaje codificado RLE: "+codificador.getMensajeCodificado());
    System.out.println("Mensaje decodificado RLE: "+codificador2.getMensajeOriginal());
    assertEquals(codificador2.getMensajeOriginal(),im.getBinInformation());

}
```

Figura 22. Prueba decodificación RLE.

```
Mensaje decodificado RLE: 423,648,-1,-1,-1,-1,-263173,-263173,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-263173,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
```

Figura 23. Impresión en pantalla prueba decodificación RLE.

Podemos observar que el decodificador de RLE nos devuelve toda la secuencia original, Para poder observar esto mejor realizamos otro test con texto para ver mejor el funcionamiento del algoritmo.

```
@Test
public void testRLEAlgorith2() {
    Scene3();
    codificador = new Codificacion(mensaje, 0);
    codificador.codificarRLE();
    System.out.println("\n\n////////////////////////////////////////// TEST RLE 2 ////////////////////////");
    System.out.println("Mensaje codificado RLE: "+codificador.getMensajeCodificado());
    mensaje = codificador.getMensajeCodificado();
    codificador2 = new Codificacion(mensaje, 1);
    codificador2.decodificarRLE();
    System.out.println("Mensaje decodificado RLE:"+codificador2.getMensajeOriginal());
}
```

Figura 24. Prueba 2 decodificación RLE.

```
Mensaje Original: a,a,a,a,a,a,e, ,l,a,s, ,r,e,p,e,t,i,c,i,o,n,e,s, ,e,n, ,m,i,s, ,t,r,a,d,u,c,c,i,o,n,e,s


////////////////////////////////////////// TEST RLE 2 //////////////////////////////////////////
Mensaje codificado RLE: 6,a;1,e;1, ;1,l;1,a;1,s;1, ;1,r;1,e;1,p;1,e;1,t;1,i;1,c;1,i;1,o;1,n;1,e;1,s;1, ;1,e;1,n;1, ;1,m;1,i;1,s;1, ;1,t;1,r;1,
Mensaje decodificado RLE:a,a,a,a,a,a,e, ,l,a,s, ,r,e,p,e,t,i,c,i,o,n,e,s, ,e,n, ,m,i,s, ,t,r,a,d,u,c,c,i,o,n,e,s,
```

Figura 25. Impresión en pantalla prueba 2.

Se puede observar que el algoritmo devuelve la misma secuencia que fue ingresada, es decir el algoritmo de decodificación de RLE funciona correctamente.

c. INFORMACIÓN DE LA CODIFICACIÓN DE FUENTE:

i. Calcular y mostrar cuántos bits tiene la información original de la imagen.

A través del atributo creado en la figura . Se puede obtener el número de bits que tiene la información original de la imagen.

ii. Calcular y mostrar cuántos bits tiene la información codificada de la imagen.

A través del atributo creado en la figura . Se puede obtener el número de bits que tiene la información codificada de la imagen.

iii. Calcular y mostrar cual es la relación de compresión.

Para el cálculo de la relación de compresión para la imagen se realiza de la misma manera como se evidencia en la figura .