



UNIVERSIDAD DIEGO PORTALES  
ESCUELA DE INFORMÁTICA &  
TELECOMUNICACIONES

## ESTRUCTURAS DE DATOS & ANÁLISIS DE ALGORITMOS

---

# Laboratorio 1: Cifrado Vigenere

---

*Autores:*  
*Narú Tapia*  
*Óscar Rodríguez*

*Profesor:*  
*Marcos Fantoal*

9 de abril 2025

---

# Índice

<b>1. Introducción.</b>	<b>2</b>
<b>2. Implementación.</b>	<b>2</b>
2.1. Atributos de la clase: . . . . .	2
2.2. Constructores: . . . . .	2
<b>3. Métodos.</b>	<b>3</b>
3.1. generarMatriz() . . . . .	3
3.2. extenderClave(String message) . . . . .	3
3.3. encrypt(String message) . . . . .	5
3.4. decrypt(String encryptedMessage) . . . . .	6
3.5. reEncrypt() . . . . .	6
3.6. public char search(int position) . . . . .	6
3.7. public char optimalSearch(int position) . . . . .	7
<b>4. Experimentación y Resultados</b>	<b>8</b>
<b>5. Conclusiones</b>	<b>11</b>
5.1. Link Github: <a href="https://github.com/O-S-C-A-T/Laboratorio1">https://github.com/O-S-C-A-T/Laboratorio1</a> . . . . .	12

---

## 1. Introducción.

El primer laboratorio consiste en crear un programa que permita crear y manejar cifrados Vigenere en JAVA para una empresa. Se solicita crear una clase llamada BigVigenere, que contiene métodos necesarios para poder realizar operaciones de cifrado Vigenere, sea encriptar, desencriptar, reencriptar, o hasta buscar un carácter según su posición/índice. Se pide evaluar con valores grandes (el mensaje a cifrar debe tener 10000 caracteres) y probar su eficiencia al cambiar el tamaño de la clave.

## 2. Implementación.

La implementación de la clase BigVigenere siguió un conjunto de pasos para poder crearse:

### 2.1. Atributos de la clase:

`int[] key`: Guarda la versión extendida y numerica de la clave.

`char[][] alphabet`: Matriz de 64x64 correspondiente al alfabeto usado en el cifrado.

`String clave`: Guarda la clave original ingresada por el usuario.

### 2.2. Constructores:

`public BigVigenere()`: Solicita al usuario la clave y luego genera la matriz del alfabeto, para este caso se implementó un método auxiliar llamado `generarMatriz()`, meramente para que se vea más ordenado, pero es un método al que solo se llama en los constructores.

`public BigVigenere(numericKey)`: Constructor que recibe una clave y la almacena, al igual que el otro constructor, llama a la función para generar la matriz, su implementación se explicará más adelante

---

## 3. Métodos.

### 3.1. generarMatriz()

Método que genera la matriz de Vigenere de 64x64 utilizando los caracteres definidos en el alfabeto personalizado (a - z, A - Z, 0 - 9, incluyendo ñ y Ñ), este método es muy importante ya que sin la matriz de Vigenere no se podría hacer nada, se implementó de la siguiente manera:

1. Primero se crea un String que contiene los 64 caracteres necesarios.
2. Se define una Array List llamada alfabeto2 que será útil para este proceso.
3. Se crea un for que se repite 64 veces (total de caracteres), el cual llenará cada índice de la Array List con cada carácter individual del String, lo cuál será útil para poder manejar cada uno con más facilidad.
4. Se crea un for (j) dentro de otro (i) para llenar nuestra matriz, ambos se repiten 64 veces. El for interior llenará la matriz de la siguiente forma:

```
this.alfabet[i][j] = alfabeto2.get(j)
```

lo cuál hará que se llene la primera columna con los 64 caracteres necesarios, es decir a la a al 9, ahí cierra el for con respecto a j, y aún dentro del for con respecto a i se implementa lo siguiente:

```
char primero = alfabeto2.get(0);  
alfabeto2.remove(0);  
alfabeto2.add(primer);
```

Se crea una variable temporal de tipo char para almacenar el primer carácter, en este caso "a", luego se elimina ese carácter de la Array List y se vuelve a introducir en esta, quedando al final, lo que haría que ahora la Array List empiece desde la letra b, y termine con la a. Se repite este proceso las veces necesarias y la matriz ya está lista y de la forma que se ha pedido.

### 3.2. extenderClave(String message)

Este método recibe el mensaje que se está utilizando, lo cual es muy importante ya que para el cifrado Vigenere se necesita que la clave tenga la misma cantidad de caracteres que el mensaje, por lo cual no podemos extender la clave sin saber la cantidad de caracteres del mensaje. Para extender la clave, se repiten sus mismos caracteres en orden hasta igualar el tamaño del mensaje, por ejemplo, si se quiere encriptar el mensaje "HOLA MUNDO", con la clave de encriptación siendo "ADIOS", resultaría de la siguiente manera:

---

Mensaje:            H O L A       M U N D O  
Clave extendida: A D I O       S A D I O

También en este caso se pide que la clave sea de la forma numérica, ya que se almacena en el atributo `int key[]`, esto se hace cambiando los caracteres por su índice respectivo en nuestro alfabeto, por ejemplo, si nuestra clave fuera "abc789", debería quedar de la siguiente forma:

Clave original:            a b c 7 8 9  
Tranformada a numérica:    0 1 2 61 62 63

En este caso es importante recordar que una vez definido un arreglo no es posible modificar su tamaño, por lo que transformarla antes de extenderla no sería muy conveniente, y para extenderla necesitamos la información del mensaje, por lo que este método hace las dos cosas en una, y lo hace de la siguiente forma:

1. Primero se define el tamaño del arreglo que almacenará la clave de la siguiente manera:

```
int[] keyExtendida = new int[message.length()];
```

2. Luego se implemente un doble for, el exterior con respecto a `i` que se repetirá tantas veces como caracteres tenga el mensaje (`message.length()`), y el interior con respecto a `j` que se repetirá 64 veces, ya que recorrerá nuestro alfabeto.

- Dentro del for `i` y antes del for `j` se inicializará una variable temporal de tipo `char` llamada "`c`":

```
char c = clave.charAt(i%Largo);
```

Lo cual de la el valor de un carácter de la clave en cierto índice, el cuál es determinado por `i%Largo` (siento `Largo` la longitud de la clave), lo cuál hace exactamente lo que se necesita, que una vez igualado su propio tamaño se repita, por ejemplo, si `Largo` fuera 3, el índice sería:

Operación	Índice
<code>0 %3</code>	0
<code>1 %3</code>	1
<code>2 %3</code>	2
<code>3 %3</code>	0
<code>4 %3</code>	1
<code>5 %3</code>	2
<code>6 %3</code>	0

Y así haciendo exactamente lo que necesitamos para extenderla.

- 
- Una vez teniendo `c` su valor se inicia el segundo `for`, el cuál recorre nuestro alfabeto hasta encontrar el carácter que sea igual a `c`, entonces añade ese índice `j` a la posición `i` del arreglo.
  - Se repite todo el proceso las veces necesarias, de esta manera extendiendo la clave y transformandola al mismo tiempo.

### 3.3. `encrypt(String message)`

Método que recibe el mensaje y lo encripta utilizando la clave almacenada, para hacerlo sigue los siguientes pasos:

1. Primero, necesitamos la clave lista para usar, es decir extendida y numérica, y como ya se tiene el mensaje, se empieza llamando al método `"extenderClave(String message)"` de la siguiente manera:

```
this.key = extenderClave(message);
```

Por lo que la clave ya estaría lista y dentro del arreglo como se pidió.

2. Luego, se crea una variable tipo `StringBuilder` denominada `"cifrado"`, la que luego se transformará en el mensaje encriptado.
3. Seguido a esto, se inicializa un `loop` de tipo `For`, cuya condición para seguir iterándose es que la variable `"i"` sea menor a la longitud del mensaje a encriptar.
4. Se crean cuatro nuevas variables:

- Una variable de tipo `Char` denominada `"c"`, equivalente al caracter en la posición `"i"` del mensaje a encriptar.
- Una variable de tipo `Int` denominada `"letraIndice"`, equivalente al índice del caracter `"c"`.
- Una variable de tipo `Int` denominada `"claveIndice"`, equivalente al elemento ubicado en `key[i]`.
- Una variable de tipo `Char` denominada `"letraCifrado"`, equivalente al caracter ubicado en la posición `[claveIndice][letraIndice]` de la matriz bidimensional de caracteres usada para el cifrado Vigenere, donde `claveIndice` es la fila del caracter, y `letraIndice` la columna de este.

5. Luego, se abre un `If statement`, cuya condición es que `letraIndice` exista (`letraIndice != -1`). Si es que existe, se agrega a la `String` del `StringBuilder` `"cifrado"` el carácter con la posición:

```
alphabet[claveIndice][letraIndice];
```

- 
6. Luego de ese If, se crea un **else if**, cuya condición es que letraIndice sea igual a -1 (que no existe). En este caso, se agrega a "cifrado" el caracter "c" sin encriptar.

```
public String encrypt(String message){ 2usages new*
    this.key = extenderClave(message);
    System.out.println(Arrays.toString(this.key));
    StringBuilder cifrado = new StringBuilder();

    for(int i=0; i<message.length(); i++){
        char c = message.charAt(i);
        int letraIndice = buscarIndice(c);

        if(letraIndice != -1){
            int claveIndice = this.key[i];
            char letraCifrada = alphabet[claveIndice][letraIndice];
            cifrado.append(letraCifrada);
        } else if (letraIndice == -1){
            cifrado.append(c);
        }
    }
    return cifrado.toString();
}
```

Figura 1: Funcion encrypt(String message)

### 3.4. decrypt(String encryptedMessage)

La funcion decrypt, como el nombre implica, se utiliza para desencriptar un mensaje encriptado, trabaja prácticamente de la misma manera que el método encrypt pero al revés, al igual que este llama al método **extenderClave()**, y utiliza el mismo método del char c que se explicó con anterioridad, se busca en el alfabeto para obtener el índice, el cuál vendría a ser el índice de la letra original, y con esto podemos ir metiendo los caracteres originales al StringBuilder y obtener el mensaje original.

### 3.5. reEncrypt()

La función reEncrypt() actua de una forma interesante.

1. Primero, se solicita al usuario un mensaje encriptado.
2. Luego, se desencripta el mensaje, haciendo uso de la funcion "decrypt" mencionada anteriormente, y se imprime a consola.
3. Luego de desencriptarlo, se pide al usuario una nueva clave de encriptación. Esta clave sobrescribe la llave de encriptación.
4. Finalmente, se encripta el mensaje previamente desencriptado con la nueva clave de encriptación, y se imprime el nuevo mensaje encriptado.

### 3.6. public char search(int position)

La función "search" toma como parámetro un entero (correspondiente al indice de un carácter en la matriz Vigenere), y busca el carácter ubicado en dicha posición.

---

Para esto, se crea una variable de tipo `Int` denominada "índice", que empieza en 0.

Además de esto, se utiliza algo conocido como "for anidado", que es un ciclo "for" dentro de otro ciclo "for", ambos cuya condición para reiterarse es que la variable que se incrementa por iteración sea menor a 64 (que es la cantidad de caracteres que tiene el alfabeto utilizado para la encriptación Vigenere).

### 3.7. `public char optimalSearch(int position)`

Un método mucho más optimizado que `search(position)`, éste método hace lo siguiente:

1. Revisa si el índice existe. Si es -1, significa que no existe, por lo que devuelve un error. Si es distinto a -1, significa que existe, por lo que continua al paso 2.
2. Crea dos variables: "fila" y "columna", donde:
  - `Fila = position / 64`
  - `Columna = position % 64`
3. Finalmente, retorna el valor con la posición `alphabet[fila][columna]`.

La optimización de éste método surge debido a que no requiere ciclos, por lo tanto, su complejidad es  $O(1)$ , la más rápida. Ésa fórmula para transformar un índice unidimensional a un índice de matriz funciona debido a que la fórmula entregada es la inversa de la utilizada para transformar un índice de matriz de 2 dimensiones a un índice unidimensional.

```
"Imaginemos que la matriz es una cuadrícula.  
Para llegar a la coordenada (x, y),  
se debe subir "y" veces y mover "x" veces hacia la derecha.  
Cada vez que se sube una vez en "y", se recorre el ancho de la  
cuadrícula completa, por lo tanto:
```

```
índice = (y * ancho) + x "
```

Debido a que la matriz es de `[63][63]`, el índice mínimo sería 0, que es el elemento en `[0][0]`, y el índice máximo que aún se mantiene dentro de la matriz es el de 4095. Entonces, si nuestro índice es 4095, se cumple lo siguiente:

- La fila en la que se encuentra es el resultado entero de la división entre el índice y el número de columnas que tiene (podemos imaginarlo como la altura, o sea, la coordenada Y).
- La columna en la que se encuentra es el módulo de la división realizada previamente, o sea, el resto de la división (que vendría a ser cuantas veces avanzó en el eje X luego de subir de fila por última vez).



---

Para el ejemplo donde el índice es 4095, se tiene que:

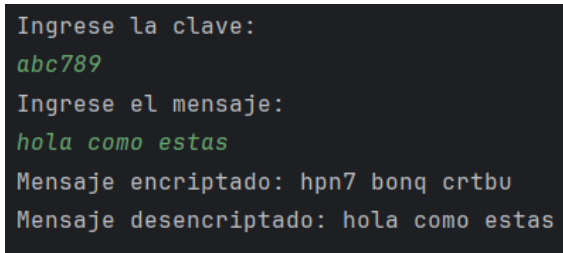
- $4095 / 64 = 63$ ;
- $4095 \% 64 = 63$ ;

La razón por la que el índice máximo es  $[63][63]$  aunque la matriz sea de orden 64 es debido a que los índices de las filas y de las columnas comienzan desde el 0, por lo tanto, el máximo sería  $([64*64] - 1)$ , que es 4095 (el -1 es simplemente porque se comienza desde el 0).

Cabe mencionar que hay métodos en Main que solo son de prueba, siendo "generarMensaje(int largo)" y "generarClaveAleatoria(int largo)", cuyos nombres son iguales que sus propósitos. Estos métodos solo se utilizan para generar aleatoriamente esas variables, con el propósito de no tener que entregar parámetros largos por consola, pero éstos son opcionales, y no necesarios para que el código funcione.

## 4. Experimentación y Resultados

Durante la realización del código, se realizaron varias pruebas, aquí un ejemplo del funcionamiento de los métodos encrypt y decrypt:



```
Ingrese la clave:
abc789
Ingrese el mensaje:
hola como estas
Mensaje encriptado: hpn7 bonq crtbu
Mensaje desencriptado: hola como estas
```

Figura 2: Prueba encrypt y decrypt.

Podemos ver en este ejemplo su correcto funcionamiento, encriptado y desencriptando devolviendo el mismo mensaje original, luego hacemos una prueba del método reEncrypt:

---

```
Ingrese mensaje encriptado:
hpn7 bonq crtbu
hola como estas
Ingrese la nueva clave:
sdBC56
zrMC 8HoP 9oMdT
```

Figura 3: Prueba reEncrypt.

Luego se realizó el ejemplo que se había pedido, el cuál consiste en introducir un mensaje con 10000 caracteres y probar la encriptación y desencriptación con diferentes claves de tamaño  $L = \{10, 50, 100, 500, 1000, 5000\}$ , y evaluar los tiempos de ejecución de cada uno, para esto se utilizó el método `nanoTime()`, y la clase `Random` para generar el mensaje y claves necesarios para el ejemplo, se realizó para todos los casos y este fue el resultado que entregó, siendo longitud de la clave, milisegundos al encriptar, y milisegundos al desencriptar respectivamente:

```
10,7,6
50,2,2
100,3,0
500,2,11
1000,2,11
5000,2,0
```

Figura 4: Tiempos de ejecución.

Se utilizaron estos datos para crear el siguiente gráfico:

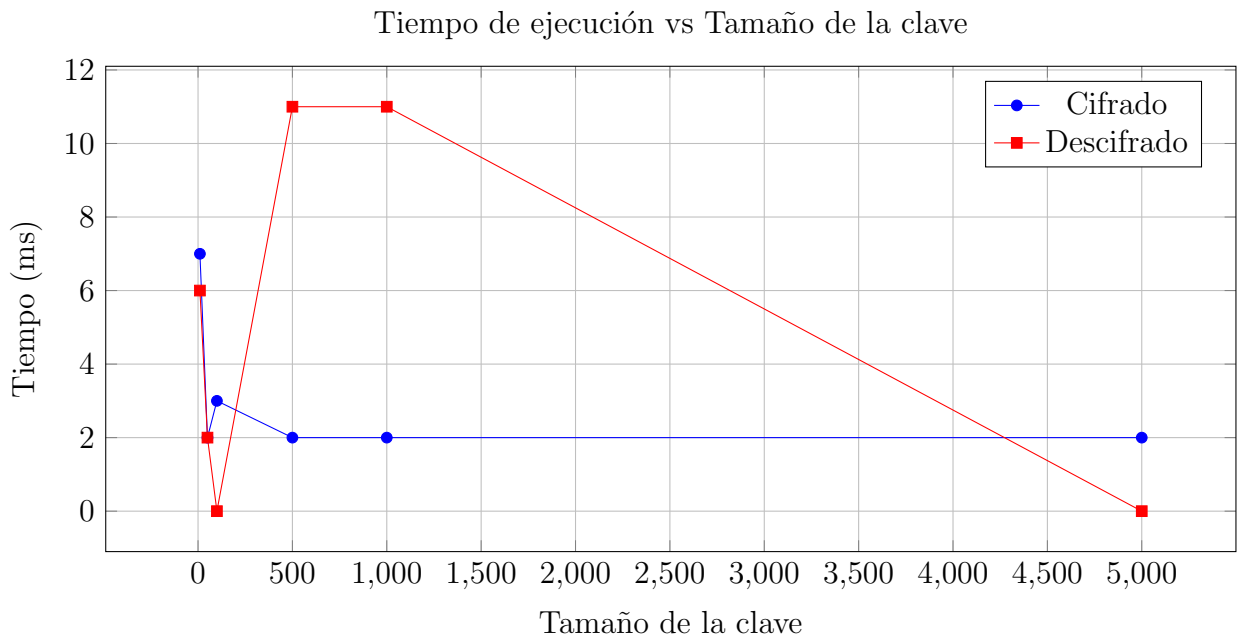


Figura 5: Tiempos de ejecución en milisegundos con diferentes claves.

Analizando los resultados, se observa que los tiempos se mantienen bastante bajos, lo que demuestra que la implementación fue eficiente, incluso con claves y mensajes largos. No hay aumentos drásticos al usar claves grandes ni al cambiarlas, lo cuál indica que el tiempo es lineal con respecto al tamaño del mensaje y no el de la clave, por lo que técnicamente la complejidad máxima de la clase es de  $O(n)$ .

---

## 5. Conclusiones

El primer laboratorio de Estructuras de Datos y Algoritmos fue bastante desafiante, ya que requirió aprender muchos conceptos para entender el cifrado Vigenere, con el propósito de crear un código para crear y revertir cifrados de este estilo. Además, fue necesario entender cómo traspasar las ideas de respuestas a código ejecutable, y fué de suma importancia aprender sintaxis de JAVA, y comandos para la redacción en  $\text{\LaTeX}$ .

Preguntas de conclusión:

¿Cuales dificultades se presentaron y cómo se resolvió?

- En el proceso de creación del código para el laboratorio, hubieron incontables dificultades. Entre las más graves:

1. El cómo llenar la matriz tomó mucho tiempo en lograrse correctamente, ya que la idea de cómo hacerlo no estaba claro. La solución para esto fué crear un alfabeto nuevo temporal, tomar el primer carácter de ese alfabeto, almacenarlo en una variable tipo Char auxiliar, se remueve del inicio y se manda al final del alfabeto temporal.
2. La función `"decrypt(String encryptedMessage)"` fué difícil de implementar, debido a que la idea que se tenía para descryptar cambiaba constantemente. Hasta que se llegó a la siguiente idea:

"Sabiedo que la clave de encripción corresponde a la fila en la que se encuentra el caracter encriptado, busca en qué columna se encuentra el carácter encriptado, y luego devuelve el carácter ubicado en esa columna, pero en la primera fila (`caracter[0][c]`)".

3.  $\text{\LaTeX}$  tiene ciertas características que dificultan escribir documentos en el, como que el caracter `' "` cambia ciertas letras, o que el simbolo `"%"` actúa ignorando todo lo que viene después de éste. La solución para ésto fue aprender a usar  $\text{\LaTeX}$  con guías, o preguntarle a una IA si el problema era menor.
- La optimización del método `"search"` fué explicada anteriormente, pero fué utilizar la inversa de la fórmula utilizada para transformar una posición de una matriz bidimensional a un índice unidimensional.
  - En término de la creación de la clave, unas recomendaciones posibles serían:
    1. Que la clave sea de igual o mayor longitud que la del mensaje a encriptar haría más difícil encontrar la clave original con el método de Kasiski (un método de ataque criptográfico que consiste en buscar palabras repetidas dentro del texto encriptado),

- 
2. Utilizar caracteres aleatorios aumenta la dificultad de descrición.
  3. Evitar utilizar claves fáciles de descifrar, que son utilizadas comúnmente, o que se repitan. (ej: "aaaa", "abc123").

**5.1. Link Github: <https://github.com/O-S-C-A-T/Laboratorio1>**