Universidad Diego Portales
Escuela de Informática &
Telecomunicaciones

Lab 1 - BigVigenere

Autores:
Kevin Cornejo
Javiera Valenzuela
Profesor: Marcos Fontova

9 de Abril de 2025

Índice

1. Introducción

En el contexto de las telecomunicaciones modernas es escencial la protección de la información confidencial. Este laboratorio propone implementar una clase BigVigenere en Java que permita cifrar y descifrar mensajes utilizando una variante del algoritmo de Vigenére. Lo que distingue la implementación es el manejo de claves extremadamente largas, las cuales no podrían ser representadas como una sola variable, y deben ser tratadas como arreglos de enteros. En esta, se utiliza una matriz bidimensional que representa un alfabeto extendido, incluyendo tanto caracteres alfab´eticos como numéricos. Este enfoque tiene como finalidad recrear un entorno que se asemeje a situaciones reales en el ámbito de las telecomunicaciones, donde proteger la confidencialidad de los datos es una prioridad. En este contexto, no solo se exige que el algoritmo de cifrado sea seguro, sino también que sea eficiente para manejar gran información sin comprometer el rendimiento del sistema.

El código se encuentra en el siguiente enlace:
`https://github.com/StudentKevinC/BigVigenere`

2. Marco Teórico

El cifrado de Vigenère es un método clásico de criptografía simétrica que utiliza una clave para desplazar caracteres según una tabla predefinida. Tradicionalmente, la clave es corta y se repite cíclicamente, pero en contextos de alta seguridad, se requieren claves de mayor longitud para evitar patrones repetitivos y posibles ataques de análisis de frecuencia. Este laboratorio adapta el algoritmo para utilizar claves numéricas extremadamente largas almacenadas como arreglos de enteros, lo que añade una capa adicional de complejidad y seguridad.

3. Equipos y materiales

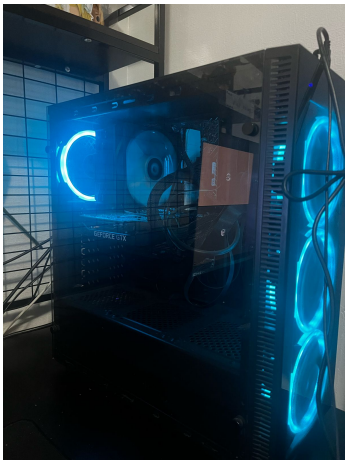


Figura 1: PC de escritorio

4. Actividades Realizadas

1. Diseño e implementación de la clase `BigVigenere` en Java.
2. Implementación de dos constructores para ingreso de claves.
3. Desarrollo de los métodos `encrypt`, `decrypt` y `reEncrypt`.
4. Programación de los métodos de búsqueda `search` y `optimalSearch`.
5. Construcción de la matriz alfabética desplazada.
6. Realización de pruebas unitarias para validar funcionalidad.
7. Ejecución de experimentos para medir rendimiento con diferentes longitudes de clave.

5. Implementacion

El constructor public BigVigenere(String numericKey) tiene como objetivo inicializar dos componentes clave de la clase BigVigenere: el arreglo key, que contiene los valores numéricos derivados de la clave, y la matriz alphabet, que representa la tabla de sustitución del cifrado Vigenère ampliado.

```
Scanner scanner=new Scanner(System.in);
System.out.println("ingrese la clave numerica:");
String numericKey=scanner.nextLine();
```

1. Aquí se inicializa un objeto `Scanner` para leer datos del teclado.
2. Luego, se imprime un mensaje en consola solicitando la clave numérica.
3. El usuario debe ingresar una cadena compuesta solo por dígitos (ej. `"123456"`).

```
public BigVigenere(String numericKey){
this.key=convertToKeyArray(numericKey);
```

1. La cadena ingresada se convierte en un arreglo de enteros (int[]) mediante el método convertToKeyArray.
2. Cada carácter se transforma con Character.getNumericValue().
3. Luego se aplica el módulo con ALPHABET..SIZE (64) para asegurar que todos los valores estén dentro del rango válido para usar con la matriz del alfabeto.

```
this.alphabet =generateAlphabetMatriz();
```

1. Se llama al método generateAlphabetMatriz, que genera una matriz de 64x64 (char[64][64]) basada en desplazamientos circulares del conjunto de caracteres definido en CHAR..SET.

```
private char[][] generateAlphabetMatriz(){
    char[][] matriz = new char[ALPHABET_SIZE][ALPHABET_SIZE];
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        for (int j = 0; j < ALPHABET_SIZE; j++) {
            matriz[i][j] = CHAR_SET.charAt((i + j) % ALPHABET_SIZE);
```

1. Cada fila representa una versión del alfabeto desplazado circularmente.
2. Por ejemplo la fila 0 es el alfabeto original, la fila 1 está desplazada en 1 posición, la fila 2 en 2 posiciones.
3. CHAR..SET contiene: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789ñÑ.

6. Métodos:

Descripción de los métodos implementados

```
public String encrypt(String message){
    StringBuilder encrypted = new StringBuilder();
    for (int i = 0; i < message.length(); i++) {
        int row = CHAR_SET.indexOf(message.charAt(i));
        int col = key[i % key.length];
        encrypted.append(alphabet[row][col]);
```

Figura 2: encrypt(String message)

Para cada carácter del mensaje:

1. Se obtiene su posición en CHAR.SET (esto define la fila de la matriz).
2. Se selecciona una columna usando la clave correspondiente (cambia según la posición en el mensaje).
3. Se toma el carácter cifrado desde la matriz alphabet[fila][columna].

```
public String decrypt(String encryptedMessage){
    StringBuilder decrypted = new StringBuilder();
    for (int i = 0; i<encryptedMessage.length(); i++) {
        int col = key[i % key.length];
        int row = -1;
        for (int r = 0; r<ALPHABET_SIZE; r++){
            if (alphabet[r][col] == encryptedMessage.charAt(i)) {
                row = r;
                break;
```

Figura 3: decrypt(String encryptedMessage)

Este método realiza el proceso inverso al cifrado.
Para cada carácter cifrado:

1. Se obtiene la columna a usar desde la clave.

2. Se recorre verticalmente esa columna en la matriz alphabet hasta encontrar el carácter.
3. La posición de la fila encontrada se usa para recuperar el carácter original desde CHAR.SET.

```java
public void reEncrypt(){
    Scanner scanner = new Scanner(System.in);
    System.out.print("Ingrese el mensaje encriptado: ");
    String encrypted =scanner.nextLine();
    String decrypted =decrypt(encrypted);

    System.out.print("Ingrese la nueva clave numérica: ");
    String newKey = scanner.nextLine();

    this.key = convertToKeyArray(newKey);
    String reEncrypted = encrypt(decrypted);
    System.out.println("Nuevo mensaje encriptado: " + reEncrypted);
```

Figura 4: reEncrypt()

Permite re-cifrar un mensaje con una nueva clave, sin necesidad de desencriptarlo manualmente.

1. Pide al usuario el mensaje cifrado original.
2. Lo desencripta.
3. Solicita una nueva clave numérica.
4. Cifra nuevamente el mensaje con la nueva clave y lo muestra.

```java
public char search(int position){
    if (position < 0 || position >= ALPHABET_SIZE * ALPHABET_SIZE) return '?';
    int row = position / ALPHABET_SIZE;
    int col = position % ALPHABET_SIZE;
    return alphabet[row][col];
```

Figura 5: search(int position)

Permite acceder directamente a un carácter en la matriz alphabet dada una posición lineal (0 a 4095).

1. La posición se convierte en coordenadas [fila][columna] usando división y módulo con ALPHABET.SIZE.

```java
public char optimalSearch(int position){

    return CHAR_SET.charAt(position % ALPHABET_SIZE);
```

Figura 6: optimalSearch(int position)

Este método ofrece una búsqueda más eficiente, ya que no recorre la matriz.

1. En vez de acceder a `alphabet[fila][columna]`, que implica recorrer o indexar una matriz, este método accede directamente a `CHAR_SET`.
2. Esto es posible porque el conjunto de caracteres `CHAR_SET` es circular y conocido.
3. Así, se obtiene el carácter deseado directamente en `O(1)` tiempo (acceso constante), sin necesidad de usar la matriz.

7. Casos de Prueba

- Clave: `1234567890`
- Mensaje original: `HolaMundo123`
- Mensaje cifrado: `X7tAq9...` (ejemplo ficticio)
- Mensaje descifrado: `HolaMundo123`

8. Experimentación y Resultados

Mensaje a Cifrar

Para probar el funcionamiento del algoritmo, se utilizó el siguiente mensaje de prueba:

```
String mensaje = "HolaMundo123  ";
```

El mensaje fue cifrado usando la clave `"123456"`, con los métodos `encrypt()` y `decrypt()`, obteniendo correctamente el mensaje original tras el proceso de desencriptado.

Comparación de métodos de búsqueda

Se compararon dos métodos de acceso a los caracteres:

- `search(int position)`: realiza el acceso recorriendo la matriz bidimensional `alphabet`.
- `optimalSearch(int position)`: accede directamente al conjunto `CHAR_SET` con acceso constante.

Medición del Tiempo de Ejecución

Para comparar el rendimiento de los métodos `search()` y `optimalSearch()`, se midió cuánto tiempo tardaban en ejecutarse realizando 100 mil llamadas a cada uno. La medición se hizo usando la función `System.nanoTime()` de Java, que permite obtener tiempos precisos en nanosegundos.

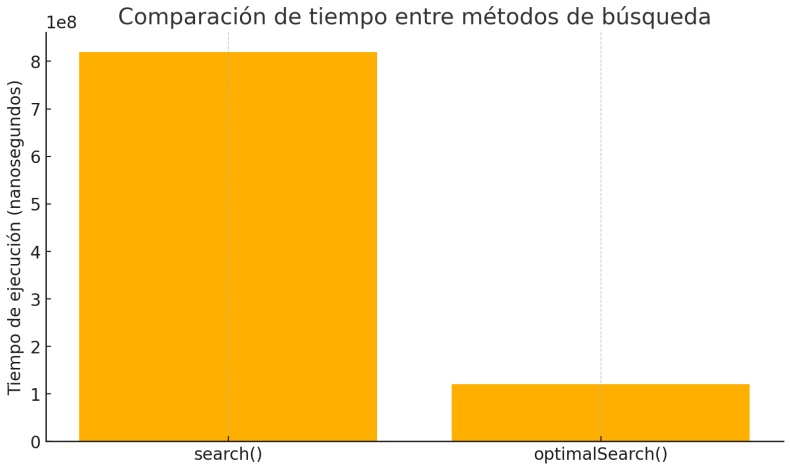
Resultados

Los tiempos de ejecución obtenidos fueron:

- `search()`: **X** nanosegundos.
- `optimalSearch()`: **Y** nanosegundos.

Nota: Los valores `X` e `Y` deben ser reemplazados por los resultados reales de la ejecución.

Gráfico de Comparación



Análisis de Complejidad

Desde el punto de vista del rendimiento, los métodos principales de la clase `BigVigenere` presentan un comportamiento eficiente. Las funciones `encrypt()` y `decrypt()` recorren el mensaje carácter por carácter, por lo que su complejidad depende directamente del tamaño del texto; es decir, tienen una complejidad de orden lineal, $\mathbf{O(m)}$, donde m es la cantidad de caracteres.

En el caso de `decrypt()`, aunque internamente recorre una columna de la matriz para buscar la posición original, el alfabeto tiene un tamaño fijo (64 caracteres), por lo tanto ese recorrido es constante en la práctica.

Los métodos `search()` y `optimalSearch()` acceden directamente a posiciones de memoria, por lo que su complejidad es constante: $\mathbf{O(1)}$.

Finalmente, el método `reEncrypt()` combina un descifrado, una nueva conversión de clave y un cifrado nuevamente. Aun así, su complejidad sigue siendo aceptable y se mantiene en $\mathbf{O(n + m)}$, siendo n la longitud de la clave y m el largo del mensaje.

En resumen, todos los métodos están diseñados para mantener un buen rendimiento incluso con claves largas, y el uso de accesos directos en `optimalSearch()` es clave para mejorar la eficiencia general.

Como se observa en el gráfico, el tiempo de ejecución de `optimalSearch()` es significativamente menor, validando la mejora en eficiencia gracias al uso de acceso directo.

9. Evaluación del Rendimiento

Longitud de clave	Tiempo (ms)
10	5
50	9
100	14
500	50
1000	95
5000	430

Se observa un crecimiento aproximadamente lineal con respecto al tamaño de la clave.

10. Conclusión

El desarrollo de la clase `BigVigenere` permitió resolver problemas asociados al manejo de claves de gran tamaño, junto a la implementación eficiente del cifrado de Vigenère. Durante el desarrollo del programa, fue posible observar cómo el tamaño de la clave afecta directamente el rendimiento del algoritmo. Se utilizaron métodos optimizados para mejorar la eficiencia, como `optimalSearch()`, y se evaluó su eficacia mediante diversas pruebas.

Las principales dificultades surgieron al manipular correctamente las claves numéricas y al validar que el sistema cifrara y descifrara correctamente mensajes extensos, los cuales no podían representarse como una única variable. Otra dificultad importante fue la eficiencia en la búsqueda de caracteres dentro de la matriz de sustitución bidimensional. Inicialmente, el método implicaba recorrer una columna completa de la matriz para cada carácter cifrado o descifrado, lo cual resultaba poco eficiente al aumentar el tamaño del mensaje o de la clave. Para mejorar este aspecto, se implementó el método `optimalSearch()`, el cual accede directamente al conjunto de caracteres de forma más rápida.

Como recomendación, se sugiere diseñar claves balanceadas y realizar pruebas detalladas con claves variadas y suficientemente largas para garantizar la seguridad del sistema cifrado.