

---

# Práctica 1 Fundamentos de Criptografía y Seguridad Informática: Criptografía Clásica

---

*Respuesta a las cuestiones de la práctica 1 de Fundamentos de Criptografía y Seguridad Informática.*

**GRUPO-1462**

**Ignacio Núñez**

**Nicolás Victorino**

Índice:

0. Introducción a la práctica.....	3
a. Lenguaje de programación escogido.....	3
b. Organización de la práctica.....	3
c. Código común entre los distintos programas .....	4
d. Patrones seguidos en los distintos programas.....	6
1. Sustitución Monoalfabeto .....	7
a. Método Afín .....	7
Criptoanálisis y resultados .....	10
b. Afín no trivial.....	11
2. Sustitución Polialfabeto .....	14
a. Método de Hill.....	14
b. Método de Vigenere.....	19
c. Criptoanálisis de Vigenere .....	21
c.1. Test de Kasinski.....	21
c.2. Índice de Coincidencia .....	27
3. Cifrado de flujo.....	31
4. Método de transposición.....	35
5. Producto de criptosistemas permutación.....	41

## 0. Introducción a la práctica

Antes de comenzar, cabe mencionar que en la explicación y discusión de cada método de encriptación usado se sigue una estructura común. En primer lugar, nos encontramos con el apartado “*Previo a la implementación*”, donde se comentan los diferentes puntos o funciones a desarrollar previo a la implementación del método. Después nos encontramos con el apartado “*Implementación del algoritmo*”, donde se explica la implementación del método. Finalmente, nos encontramos con el apartado “*Criptografía y resultados*”, donde se habla más a fondo de la seguridad del cifrado, y de cómo podemos ejecutarlos.

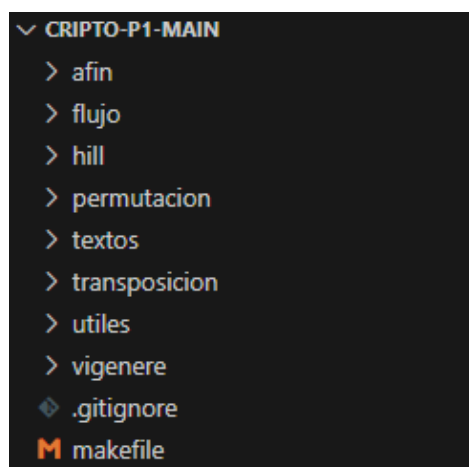
Sin embargo, en caso de no necesitarse se puede prescindir de alguno de ellos.

### a. Lenguaje de programación escogido

Para realizar esta práctica, hemos decidido hacer uso del lenguaje de programación C, debido a su eficiencia y velocidad (se compila a código máquina optimizado) y a que es un lenguaje de bajo nivel, que permite mucha flexibilidad a la hora de implementar los algoritmos criptográficos y da un mayor control sobre la gestión de la memoria.

### b. Organización de la práctica

La práctica está estructurada de la siguiente forma:



Como se puede observar, hemos decidido separar cada algoritmo de cifrado en carpetas, haciendo uso de un fichero “*utils.c*” general para todos los programas. Además, contamos con un makefile que se encarga de compilar y ejecutar todos ellos.

Además, hay 2 directorios extra:

- *Textos*: Contiene todos los textos que se usan de base para probar los algoritmos de cifrado. Además, aquí también podemos encontrar los archivos donde se guardan los textos cifrados, que por lo general se guardan en el archivo “*adios.txt*”.

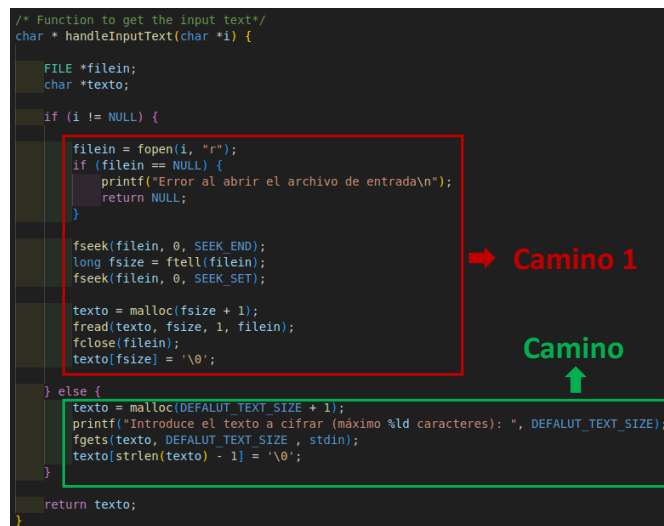
- *Obj*: Contiene todos los archivos “\*.o”, necesarios para la compilación de los distintos programas.

Por último, cabe mencionar que en la carpeta de *vigenere* también se encuentran los programas necesarios para realizar el criptoanálisis de este (“*criptoanalisisKasinski.c*” y “*criptoanalisisIndice.c*”). Además, en la carpeta de *afin* también encontramos el programa “*afin\_no\_trivial.c*”, que contiene la modificación del algoritmo de cifrado afin realizada por nosotros.

### c. Código común entre los distintos programas

La mayoría de los programas desarrollados comparten ciertas características, las cuales se han extraído en funciones (las cuales podemos encontrar en “*utils.c*”):

- Función `handleInputText()`:



```

/* Function to get the input text*/
char * handleInputText(char *i) {

    FILE *filein;
    char *texto;

    if (i != NULL) {

        filein = fopen(i, "r");
        if (filein == NULL) {
            printf("Error al abrir el archivo de entrada\n");
            return NULL;
        }

        fseek(filein, 0, SEEK_END);
        long fsize = ftell(filein);
        fseek(filein, 0, SEEK_SET);

        texto = malloc(fsize + 1);
        fread(texto, fsize, 1, filein);
        fclose(filein);
        texto[fsize] = '\0';

    } else {
        texto = malloc(DEFAULT_TEXT_SIZE + 1);
        printf("Introduce el texto a cifrar (máximo %ld caracteres): ", DEFAULT_TEXT_SIZE);
        fgets(texto, DEFAULT_TEXT_SIZE, stdin);
        texto[strlen(texto) - 1] = '\0';
    }

    return texto;
}

```

El código muestra la función `handleInputText` con dos caminos de ejecución:

- Camino 1** (rojo): Se ejecuta cuando se proporciona un archivo de entrada (`i != NULL`). Se abre el archivo, se busca el final para obtener el tamaño, se busca el inicio y se lee el contenido en un búfer de memoria.
- Camino 2** (verde): Se ejecuta cuando no se proporciona un archivo (`i == NULL`). Se solicita al usuario que introduzca el texto por entrada estándar, limitando el tamaño por la macro `DEFAULT_TEXT_SIZE`.

Esta función obtiene el texto que se quiere cifrar. Tiene 2 caminos principales a la hora de hacerlo:

- En caso de haber ejecutado el programa con el parámetro `-i`, abre el fichero proporcionado y extrae el texto (**Camino 1**).
- En caso contrario, obtiene el texto por entrada estándar, con un tamaño máximo establecido por la macro `DEFAULT_TEXT_SIZE` (**Camino 2**).

- Función `handleOutputText()`:

```
/* Function to handle the output of text*/
int handleOutputText(char *o, char *texto) {
    FILE *fileout;

    if (o != NULL) {
        fileout = fopen(o, "w");
        if (fileout == NULL) {
            printf("Error al abrir el archivo de salida\n");
            return -1;
        }
        fprintf(fileout, "%s", texto);
        fclose(fileout);
    } else {
        printf("Texto: %s\n", texto);
    }

    return 0;
}
```

*Camino* ↑

→ *Camino*

Esta función gestiona el texto una vez ha sido cifrado/descifrado. Sigue los mismos 2 caminos que la función anterior, solo que en lugar de tener en cuenta el parámetro `-i`, tiene en cuenta el `-o`.

- Función `filtrarTexto()`:

```
/* Function to preprocess the text to encode*/
void filtrarTexto(char *texto) {
    int i = 0;
    while (texto[i] != '\0') {
        if (texto[i] >= 'A' && texto[i] <= 'Z') {
            texto[i] = texto[i] + 32;
        } else if (texto[i] < 'a' || texto[i] > 'z') {
            /* Elimino el caracter*/
            int j = i;
            while (texto[j] != '\0') {
                texto[j] = texto[j + 1];
                j++;
            }
            /* Como he eliminado un caracter, tengo que mover el indice para atras*/
            i--;
        }
        i++;
    }
}
```

Esta función preprocesa el texto a cifrar. Sustituye todas las letras mayúsculas por letras minúsculas y elimina el resto de los caracteres, incluidos los espacios y saltos de línea.

#### d. Patrones seguidos en los distintos programas

- Función printExeInfo():

```
void printExeInfo() {  
    printf("./afin {-C|-D} {-m |Zm|} {-a N} {-b N+} [-i filein] [-o fileout]\n");  
    printf("-C|-D: -C para cifrar, -D para descifrar\n");  
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");  
    printf("-a N: clave de cifrado\n");  
    printf("-b N+: clave de cifrado\n");  
    printf("-i filein: archivo de entrada\n");  
    printf("-o fileout: archivo de salida\n");  
}
```

En caso de haber un fallo a la hora de ejecutar un programa, todos ellos contienen esta función, que imprime el método correcto para hacerlo. En este caso, estamos viendo a modo de ejemplo la función del programa afin.

- Función cifrar () /descifrar ():

```
void cifrar(char *texto, mpz_t m, mpz_t a, mpz_t b) { ...  
void descifrar(char *texto, mpz_t m, mpz_t a, mpz_t b) { ...
```

Todos los programas (a excepción del cifrado por transposición que solo contiene la función cifrar, lo cual explicaremos más adelante) contienen dos funciones llamadas *cifrar* y *descifrar*. Estas, como bien explica el nombre, son las encargadas de cifrar y descifrar el texto de entrada. Los parámetros de entrada de estas funciones varían dependiendo de las necesidades de cada algoritmo. En la imagen, vemos como ejemplo la cabecera de las funciones del programa de cifrado afin.

- Por último, cabe mencionar que a la hora de explicar los programas omitiremos las primeras líneas de la función main, que consisten en la obtención de los parámetros pasados por terminal necesarios para la ejecución del programa.

# 1. Sustitución Monoalfabeto

## a. Método Afín

### Previo a la implementación:

Este método hace uso de los algoritmos de Euclides (para comprobar que la clave 'a' es válida en función del tamaño de alfabeto 'm') y Euclides extendido (para conseguir el inverso multiplicativo de la clave 'a' cuando queramos descifrar el texto).

Ambas funciones hacen uso del tipo de datos "mpz\_t", incluido en la librería GMP. Podemos encontrarlas en el fichero "utils.c":

- Euclides:

```
/* Euclides loop */
while (mpz_cmpabs_ui(b, 0) != 0) {

    /* We will always save memory for one more element */
    result = (mpz_t *)realloc(result, (i+2) * sizeof(mpz_t));

    if (result == NULL) {
        printf("Error en la asignacion de memoria\n");
        mpz_clear(temp);
        mpz_clear(tempa);
        mpz_clear(tempb);
        exit(1);
    }
    mpz_init(result[i]);

    mpz_tdiv_q(result[i], a, b); → 1
    mpz_mod(temp, a, b);
    mpz_set(a, b);
    mpz_set(b, temp);

    i++;
}

/* Save the last element */
mpz_init(result[i]);
mpz_set(result[i], a); → 2

*z = i + 1;
```

Guardamos en el array 'result' los cocientes y el máximo común divisor de 'a' y 'm', siguiendo el algoritmo de Euclides. Además, escribimos en la variable 'z' el tamaño de este array.

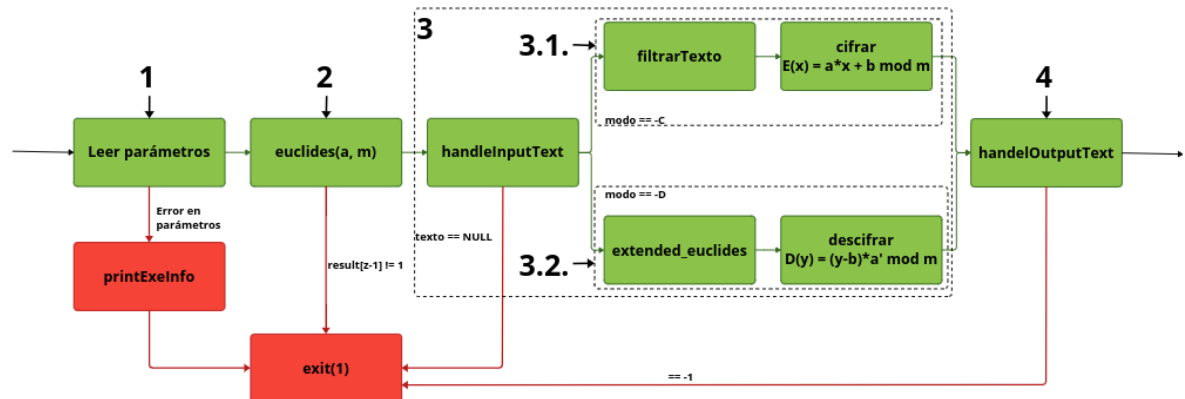
En cada iteración, reservamos espacio para 2 valores más en el array 'result'. El primero para guardar el cociente calculado en esa iteración (1), y el segundo para, en caso de ser la última iteración, tener espacio para guardar el máximo común divisor (2).

- Euclides extendido:

En esta función (*extended\_euclides*), calculamos el inverso multiplicativo de 'a' con tamaño de alfabeto 'm'. Para ello, hacemos uso de los cocientes calculados con el algoritmo de Euclides.

## Implementación del algoritmo:

Ahora, vamos a explicar paso a paso como hemos integrado este método, siguiendo el siguiente esquema:



### 1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./afin {-C|-D} {-m |Zm|} {-a N+} {-b N+} [-i filein]
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-a N+: clave de cifrado\n");
    printf("-b N+: clave de cifrado\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Hemos decidido mantener los parámetros de entrada propuestos por el enunciado. Como se puede ver, contamos con 4 parámetros obligatorios ("-C|-D", "-m", "-a", "-b") y 2 opcionales ("-i", "-o").

2. El siguiente paso es comprobar la validez de la clave introducida ('a' y 'm' principalmente). Se tiene que cumplir que estos 2 parámetros sean coprimos, para lo que haremos uso del algoritmo de Euclides mencionado previamente.

```
/* Compruebo que a y b se pueden usar como claves con m*/
result = euclides(a, m, &z);

if (mpz_cmp_ui(result[z - 1], 1) != 0) {
    printf("La clave a no es válida\n");
    /* Libero memoria*/
    for (int j = 0; j <= z; j++) {
        mpz_clear(result[j]);
    }
    free(result);
    mpz_clear(m);
    mpz_clear(a);
    mpz_clear(b);
    exit(1);
}
```



3. Procedemos a leer el texto de entrada y desciframos o ciframos el texto en función del parámetro introducido:

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}

/* Cifrar o descifrar texto*/
if (strcmp(modo, "-C") == 0) {
    filtrarTexto(texto);
    cifrar(texto, m, a, b);
} else {
    descifrar(texto, m, a, b);
}

/* Escribir nuevo texto en salida*/
if (handleOutputText(o, texto) == -1) {
    exit(1);
}
```

### 3.1. Caso de cifrado:

Primero, debemos filtrar el texto de entrada con la función “*filtrarTexto*” mencionada previamente. Una vez realizado este paso, procedemos a cifrar el texto mediante la fórmula explicada en teoría, aplicándosela a cada carácter del texto.

$$E(x) = a \times x + b \bmod m$$

### 3.2. Caso de descifrado:

En este caso no es necesario filtrar el texto, ya que en principio este paso ya se ha realizado a la hora de cifrarlo. Lo que si es necesario es calcular el inverso multiplicativo de ‘a’ respecto a ‘m’, para lo que haremos uso del algoritmo de Euclides extendido.

Una vez obtenido, podemos descifrar el texto mediante la fórmula explicada en teoría, aplicándosela a cada carácter del texto.

$$D(y) = (y - b) \times a^{-1} \bmod m$$

4. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

```
/* Escribir nuevo texto en salida*/
if (handleOutputText(o, texto) == -1) {
    exit(1);
}
```

## **Criptoanálisis y resultados**

Para probar este método, podemos hacer uso del makefile ('make run\_afin\_C' para cifrar, 'make run\_afin\_D' para descifrar). Por defecto, estos comandos utilizan el archivo de entrada 'hola.txt', y 'adios.txt' como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta 'textos' si se desea, o probar con textos propios.

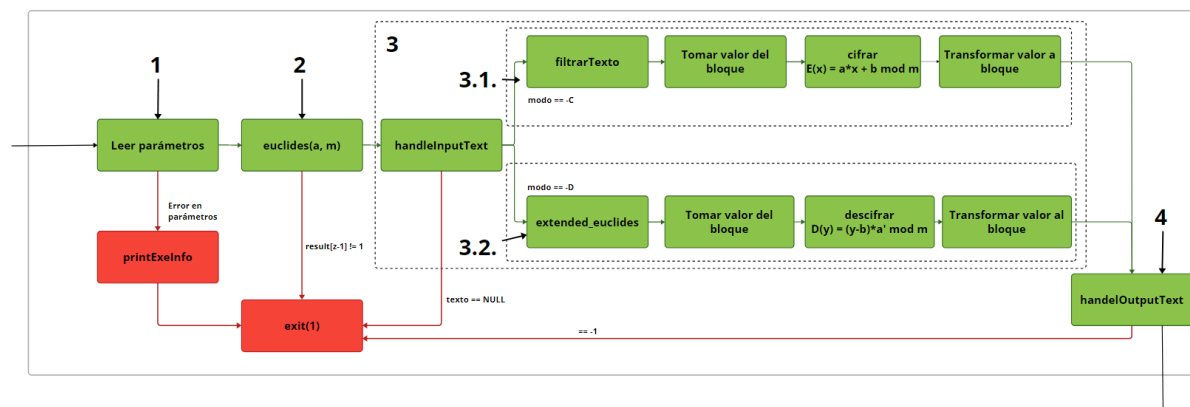
## b. Afín no trivial

### Previo a la implementación:

El método afín contiene un problema importante, y es que, manteniendo la base explicada en el apartado anterior, para textos planos en inglés donde se entiende que 'm' toma el valor de 26, existen muy pocas combinaciones de claves. Esto ocurre porque 'a' y 'm' deben ser primos entre sí y 'a' debe ser menor que 'm'. Esto último no es que sea matemáticamente estrictamente necesario, pero si se diese el caso existiría una clave 'a<sub>x</sub>' menor que 'm' que daría como resultado el mismo texto resultante. Por su parte, 'b' tiene mayor libertad de valores, pero también está limitado a ser menor que 'm' por el mismo motivo. Por ello, descifrar un texto "por fuerza bruta" es relativamente sencillo y no lleva mucho tiempo computacionalmente.

Para solucionar esto debemos idear una forma de aumentar la posibilidad de claves y esto pasa por aumentar el valor de 'm' que claramente nos está lastrando. Para ello la idea será en lugar de usar un carácter por valor, usar varios. De esta forma, la cantidad de posibles valores será de  $26^n$  donde 'n' es el tamaño del bloque de caracteres. Así, el número de posibles claves se eleva exponencialmente. Por temas de simplificación para la programación, elaboraremos el método con  $n=2$ , pero es perfectamente posible realizarse para valores de n mayores.

### Implementación del algoritmo



#### 1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./afin_no_trivial {-C|-D} {-m |Zm|} {-a N} {-b N+} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-a N: clave de cifrado\n");
    printf("-b N+: clave de cifrado\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Idénticamente al método afín estándar, usaremos los parámetros propuestos del enunciado para la ejecución del programa.

2. Tras comprobar que los parámetros son correctos y adecuados comprobaremos si las claves son válidas. Para ello, usaremos de nuevo el algoritmo de Euclides y si 'a' y 'm' no son primos pararemos la ejecución. Recordemos que para este método estamos usando la librería GMP con sus correspondientes variables y funciones.

```
/* Compruebo que a y b se pueden usar como claves*/
result = euclides(a, m, &z);

if (mpz_cmp_ui(result[z - 1], 1) != 0) {
    printf("La clave a no es válida\n");
    /* Libero memoria*/
    for (int j = 0; j <= z; j++) {
        mpz_clear(result[j]);
    }
    free(result);
    mpz_clear(m);
    mpz_clear(a);
    mpz_clear(b);
    exit(1);
}
```

3. Ahora filtraremos el texto para escoger solo los caracteres que necesitemos para posteriormente cifrar o descifrarlo. Es importante mencionar que para cuando se cifre es posible que se añada un carácter al texto. Esto se debe a que como vamos a cifrar los bloques de dos en dos, necesitamos que el número de caracteres sea par. Usaremos la función `addFilling()` para rellenar el texto con el carácter.

```
/* Comprobamos si la longitud del texto es par */
if (len % 2 != 0) {
    texto = addFilling(2, texto);
    len = strlen(texto);
}
```

3.1 Para cifrar el texto seleccionaremos los dos caracteres a cifrar y calcularemos su valor total. De igual forma que funcionan las unidades y decenas de cualquier sistema numérico, el primer carácter tendrá su valor multiplicado por 26 (valor de m) y el segundo su valor únicamente. Luego realizaremos los cálculos de la función con 'a' y 'b' y reconvertiremos el valor aplicando el mismo método, pero a la inversa. El primer carácter será el cociente de la división y el segundo el resto.

```

mpz_set_ui(temp_char1, texto[i] - 'a');
mpz_set_ui(temp_char2, texto[i + 1] - 'a');

/* Obtenemos el valor de ambos caracteres*/
mpz_mul_ui(aux, temp_char1, m); // decenas
mpz_add(aux, aux, temp_char2); // unidades

/* Multiplicamos por a */
mpz_mul(aux, aux, a);

/* Sumar b al resultado de la multiplicación*/
mpz_add(aux, aux, b);

/* Calcular el módulo m del resultado*/
mpz_mod(aux, aux, m);

/* Calculamos cual es el nuevo numero*/
mpz_tdiv_q_ui(temp_char1, aux, m); // decenas
mpz_tdiv_r_ui(temp_char2, aux, m); // unidades

```

3.2 Descifrar usa la misma lógica que el cifrado, pero de forma inversa. Lo primero que haremos será calcular el inverso multiplicativo usando Euclides extendido y haremos la transformación de ambas cifras en un único valor, calcularemos el valor y retransformaremos el valor en dos caracteres.

```

v = extended_euclides(a, m, &z);

for (i = 0; i < len; i+=2) {

    mpz_set_ui(temp_char1, texto[i] - 'a');
    mpz_set_ui(temp_char2, texto[i+1] - 'a');

    /* Obtenemos el valor de ambos caracteres*/
    mpz_mul_ui(aux, temp_char1, 26); // decenas
    mpz_add(aux, aux, temp_char2); // unidades

    /* Restamos por b */
    mpz_sub(aux, aux, b);

    /* Multiplicamos por el inverso de a */
    mpz_mul(aux, aux, v[z-1]);

    /* Calcular el módulo m del resultado*/
    mpz_mod(aux, aux, m);

    /* Calculamos cual es el nuevo numero*/
    mpz_tdiv_q_ui(temp_char1, aux, 26); // decenas
    mpz_tdiv_r_ui(temp_char2, aux, 26); // unidades
}

```

4. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

```

/* Escribir nuevo texto en salida*/
if (handleOutputText(o, texto) == -1) {
    exit(1);
}

```

## Criptografía y resultados

Con esto hemos obtenido un método de cifrado con mucha mayor fortaleza que el afín trivial. Es cierto que si hubiésemos realizado el método con tres caracteres o más el número de claves posibles sería aún mayor y por ende será aún más robusto a posibles ataques. Se puede probar la ejecución, cifrado y descifrado del algoritmo a través del comando del makefile `run_afin_no_trivial_(C/D)`.

## 2. Sustitución Polialfabeto

### a. Método de Hill

#### Previo a implementación

Para este método hemos hecho uso de los algoritmos de Euclides (para comprobar la validez de la matriz clave) y Euclides extendidos (para conseguir la matriz inversa de la clave para descifrar el texto). Sin embargo, las funciones desarrolladas para el método afín no nos sirven, ya que hacen uso de datos de tipo *'mpz\_t'*. Es por ello por lo que hemos desarrollado las funciones *'euclides2'* y *'extended\_euclides2'*, que utilizan datos de tipo entero (*int*).

Podemos encontrar ambas funciones en *"utils.c"*:

- Euclides2:

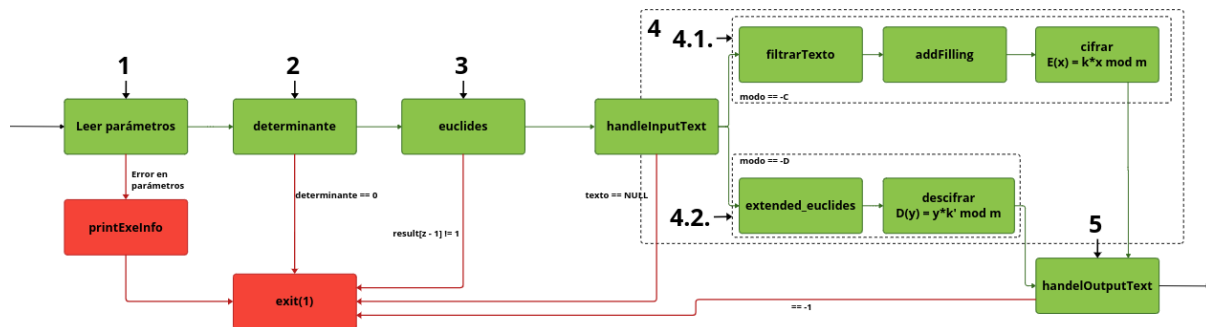
```
while (det != 0) {  
    result = (int *)realloc(result, (i+2) * sizeof(int));  
    if (result == NULL) {  
        printf("Error en la asignacion de memoria\n");  
        exit(1);  
    }  
  
    result[i] = m / det; ➡ 1  
  
    temp = m;  
    m = det;  
    det = temp % det;  
  
    i++;  
}  
  
result[i] = m; ➡ 2  
*z = i + 1;
```

Funciona exactamente igual que la función *'euclides'*, devuelve un array compuesto por los cocientes obtenidos en cada iteración (1) y el máximo común divisor de las variables *'det'* y *'m'* (2).

- Extended\_euclides2:

También funciona igual que la versión explicada en el método afín, salvo el cambio del tipo de datos. Obtenemos el inverso multiplicativo de *'det'* en función del tamaño de alfabeto *'m'*. Esto lo consigue haciendo uso de los cocientes calculados mediante la función *'euclides2'*

## Implementación del algoritmo



### 1. Parámetros necesarios para la ejecución del algoritmo:

```

void printExeInfo() {
    printf("./hill {-C|-D} {-m |Zm|} {-n Nk} {-k filek} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-n Nk: tamaño de la clave (tamaño de la matriz)\n");
    printf("-k filek: archivo donde esta guardado la matriz para cifrar o descifrar el texto\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
    
```

Hemos decidido mantener los parámetros de entrada propuestos por el enunciado. Como se puede contar con 4 parámetros obligatorios ("-C|-D", "-m", "-n", "-k") y 2 opcionales ("-i", "-o").

### 2. Calculamos el determinante de la matriz clave y validamos su validez:

```

/* Calculo el determinante de la matriz y compruebo si es valido */
if (n == 2) {
    det = keys[0][0] * keys[1][1] - keys[0][1] * keys[1][0];
} else if (n == 3) {
    det = keys[0][0] * keys[1][1] * keys[2][2] + keys[0][1] * keys[1][2] * keys[2][0] + keys[0][2] * keys[1][0] * keys[2][1] - keys[0][2] * keys[1][1] * keys[2][0] - keys[0][0] * keys[1][2] * keys[2][1] - keys[0][1] * keys[1][0] * keys[2][2];
} else {
    printf("El tamaño de la matriz de claves no es válido\n");
    for (int x = 0; x < m; x++) {
        free(keys[x]);
    }
    free(keys);
    exit(1);
}

if (det == 0) {
    printf("Error: La matriz no tiene inversa\n");
    for (int x = 0; x < m; x++) {
        free(keys[x]);
    }
    free(keys);
    exit(1);
}
    
```

Como se puede ver, solo aceptamos matrices de tamaño 2 o 3, ya que calculamos manualmente el determinante de cada uno. Además, a la hora de multiplicar matrices también realizamos la operación manualmente, llegando únicamente hasta matrices de tamaño 3.

En caso de que el determinante calculado fuera igual que 0, descartamos la matriz, ya que esto quiere decir que esta matriz no tiene inversa.

3. Aplicamos el algoritmo de Euclides al determinante de la matriz con tamaño de alfabeto 'm':

```
int * invdet = euclides2(det, m, n, &z);

if (invdet[z - 1] != 1) {
    printf("Error: La matriz no tiene inversa\n");
    free(invdet);
    for (int x = 0; x < m; x++) {
        free(keys[x]);
    }
    free(keys);
    exit(1);
}
```

4. Procedemos a leer el texto de entrada y desciframos o ciframos el texto en función del parámetro introducido:

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}

/* Cifrar o descifrar */
if (strcmp(modo, "-C") == 0) {
    filtrarTexto(texto);

    /* Anyado relleno al texto original para que su longitud sea multiplo del tamaño del array */
    int len = strlen(texto);
    int new_len = len + (n - (len % n));
    texto = realloc(texto, new_len + 1); // +1 para el carácter nulo
    if (texto == NULL) {
        printf("Error en la asignacion de memoria\n");
        exit(1);
    }

    for (int x = len; x < new_len; x++) {
        texto[x] = 'x';
    }
    texto[new_len] = '\0';
    cifrar(texto, m, n, keys);
} else {
    descifrar(texto, m, n, det, keys);
}
```

#### 4.1. Caso de cifrado:

Primero, debemos filtrar el texto de entrada con la función “*filtrarTexto*” mencionada previamente. Una vez realizado este paso, debemos comprobar si es necesario añadir relleno al texto original (en nuestro caso rellenamos con el carácter ‘x’). Esto se debe a que el tamaño de la matriz clave (‘*keys*’) debe ser divisor de la longitud del texto. De no ser así, no se podrían realizar las multiplicaciones de matrices correctamente.

Por último, procedo a cifrar el texto mediante la fórmula explicada en teoría, aplicándosela en bloques del tamaño de la matriz clave.

$$E(x) = x[1][i] \times k[i][j] \bmod m$$



#### 4.2. Caso de descifrado:

En este caso no es necesario filtrar el texto ni añadir relleno (ya se hace al cifrar el texto). Lo que si es necesario es calcular el inverso multiplicativo de la matriz clave ('keys') respecto a 'm'. Para ello, tenemos que realizar 3 pasos:

1. Obtenemos el inverso multiplicativo del determinante de la matriz clave ('keys'):

```
//Obtengo el inverso multiplicativo
int *v = extended_euclides2(det, m, n, &z);
```

2. Calculo la matriz adjunta:

```
//Calculo la matriz adjugada de la matriz de claves
int **adj = (int **)malloc(n * sizeof(int *));
for (int x = 0; x < n; x++) {
    adj[x] = (int *)malloc(n * sizeof(int));
}

if (n == 2) {
    adj[0][0] = k[1][1];
    adj[0][1] = -k[0][1];
    adj[1][0] = -k[1][0];
    adj[1][1] = k[0][0];
} else {
    adj[0][0] = k[1][1] * k[2][2] - k[1][2] * k[2][1];
    adj[0][1] = -1 * (k[1][0] * k[2][2] - k[1][2] * k[2][0]);
    adj[0][2] = k[1][0] * k[2][1] - k[1][1] * k[2][0];
    adj[1][0] = -1 * (k[0][1] * k[2][2] - k[0][2] * k[2][1]);
    adj[1][1] = k[0][0] * k[2][2] - k[0][2] * k[2][0];
    adj[1][2] = -1 * (k[0][0] * k[2][1] - k[0][1] * k[2][0]);
    adj[2][0] = k[0][1] * k[1][2] - k[0][2] * k[1][1];
    adj[2][1] = -1 * (k[0][0] * k[1][2] - k[0][2] * k[1][0]);
    adj[2][2] = k[0][0] * k[1][1] - k[0][1] * k[1][0];
}
```

3. Con la matriz adjunta y el inverso multiplicativo de la matriz clave ('v[z-1]') puedo calcular la matriz inversa:

```
//Calculo la matriz inversa
int **inv = (int **)malloc(n * sizeof(int *));
for (int x = 0; x < n; x++) {
    inv[x] = (int *)malloc(n * sizeof(int));
}

for (int x = 0; x < n; x++) {
    for (int y = 0; y < n; y++) {
        inv[x][y] = (adj[x][y] * v[z - 1]);
        inv[x][y] = inv[x][y] % m;

        if (inv[x][y] < 0) {
            inv[x][y] = m + inv[x][y];
        }
    }
}
```

Una vez obtenido, podemos descifrar el texto mediante la fórmula explicada en teoría, aplicándosela en bloques del tamaño de la matriz clave.

$$D(y) = y[1][i] \times k^{-1}[i][j] \bmod m$$

5. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función "handleOutputText", mencionada previamente.

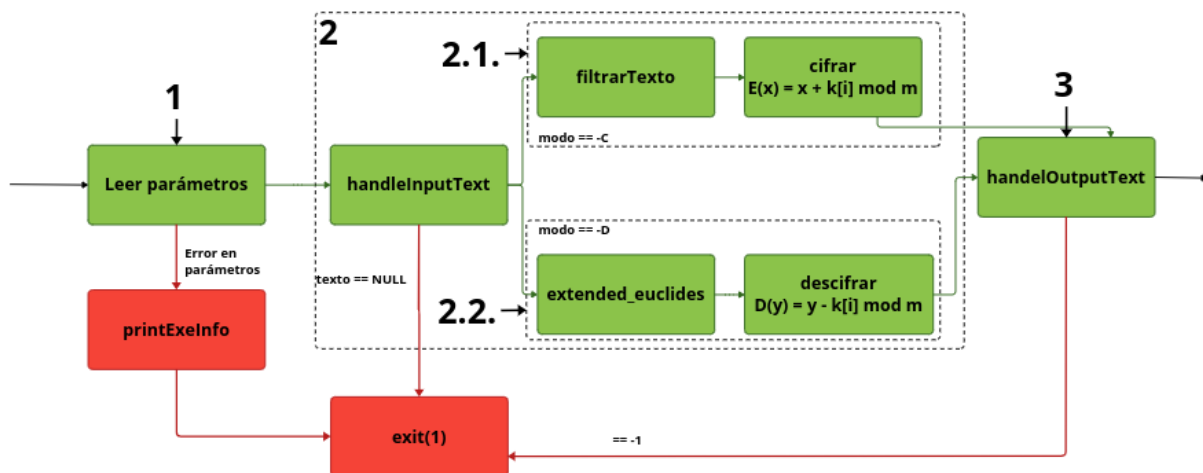
```
/* Escribir nuevo texto en salida*/  
if (handleOutputText(o, texto) == -1) {  
    exit(1);  
}
```

## Criptoanálisis y resultados

Para probar este método, podemos hacer uso del makefile ('make run\_hill\_C' para cifrar, 'make run\_hill\_D' para descifrar). Por defecto, estos comandos utilizan el archivo de entrada 'hola.txt', y 'adios.txt' como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta 'textos' si se desea, o probar con textos propios.

## b. Método de Vigenere

### Implementación del algoritmo



1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./vigenere {-C|-D} {-m |Zm|} {-k key} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-k key: clave para cifrar o descifrar el texto\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Hemos decidido mantener los parámetros de entrada propuestos por el enunciado. Como se puede contar con 3 parámetros obligatorios ("-C|-D", "-m", "-k") y 2 opcionales ("-i", "-o").

2. Procedemos a leer el texto de entrada y desciframos o ciframos el texto en función del parámetro introducido:

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}

/* Cifrar o descifrar */
if (strcmp(modo, "-C") == 0) {
    filtrarTexto(texto);
    cifrar(texto, m, k);
} else {
    descifrar(texto, m, k);
}
```

### 2.1. Caso de cifrado:

Primero, debemos filtrar el texto de entrada con la función “*filtrarTexto*” mencionada previamente. Una vez realizado este paso, procedemos a cifrar el texto mediante la fórmula explicada en teoría:

$$E(x) = (x_1 + k_1, \dots, x_n + k_n) \bmod m$$

### 2.2. Caso de descifrado:

En este caso no es necesario filtrar el texto, ya que ya se ha hecho a la hora de cifrarlo. Podemos proceder a aplicar la fórmula vista en clase sobre el texto cifrado sin ningún paso previo:

$$E(x) = (x_1 - k_1, \dots, x_n - k_n) \bmod m$$

3. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

```
/* Escribir nuevo texto en salida */  
if (handleOutputText(o, texto) == -1) {  
    exit(1);  
}
```

## Criptografía y resultados

## c. Criptoanálisis de Vigenere

### c.1. Test de Kasinski

#### Previo a implementación

Hemos desarrollado un pseudodiccionario el cual vamos a usar para realizar el criptoanálisis de Vigenere por el método del Test de Kasinski.

```
typedef struct {  
    char **cadenas;  
    long **posiciones;  
} Diccionario;
```

Este diccionario consiste en 2 arrays, el primero almacena las palabras extraídas del texto, y el segundo de arrays de enteros donde se guardan las posiciones de las palabras (limitado el tamaño de estos arrays por la macro MAX\_WORDS\_REPETITIONS).

Estos 2 arrays se relacionan por el índice, coincidiendo el índice donde se guarda la palabra con el índice donde se guarda su array de posiciones. Además, se inicializan a -1 todas sus posiciones, consiguiendo así saber dónde se ha guardado la última palabra. Además, esto ayuda a optimizar el programa, evitando realizar iteraciones de más cuando recorremos el diccionario.

Una vez creada la estructura del diccionario, hemos creado distintos métodos para trabajar con ella. Entre estos, podemos encontrar métodos para crear el diccionario (*createDiccionario()*), para eliminarlo (*freeDiccionario()*), para imprimirlo (*printDiccionario()*) y para encontrar la posición de una palabra (*findWord()*):

- CreateDiccionario():

Esta función devuelve un puntero a una estructura de tipo *'Diccionario'*, con todos sus valores inicializados a -1 en caso del array de posiciones, o a '\0' en caso del array de cadenas. Necesita 2 parámetros de entrada, *'lenc'* para indicar la longitud de las cadenas que vamos a guardar, y *'lent'* para indicar la longitud del texto que va a ser criptoanalizado.

El tamaño de los arrays para los que se reserva memoria es igual al número de cadenas distintas de longitud *'lenc'* que podrían caber en el texto con una longitud *'lent'*.

- FreeDiccionario():

Esta función libera la toda la memoria de una estructura de tipo *'Diccionario'*, dada su longitud de cadena (*'lenc'*) y su longitud de texto criptoanalizado (*'lent'*).

- PrintDiccionario():

Imprime todos los valores de un diccionario, indicando la cadena y las posiciones en las que se encuentra en el texto.

- FindWord():

Dada una cadena de caracteres, identifica si el diccionario ya la contiene, devolviendo su posición dentro del array. En caso contrario, devuelve -1.

Adicionalmente, hemos creado funciones para ayudarnos a dividir el texto en subcadenas (palabras) y guardarlas correctamente en el diccionario (*analyzeText()*), y para calcular el máximo común divisor de las distancias entre todas las posiciones en el texto de estas cadenas (*obtenerMCD()*):

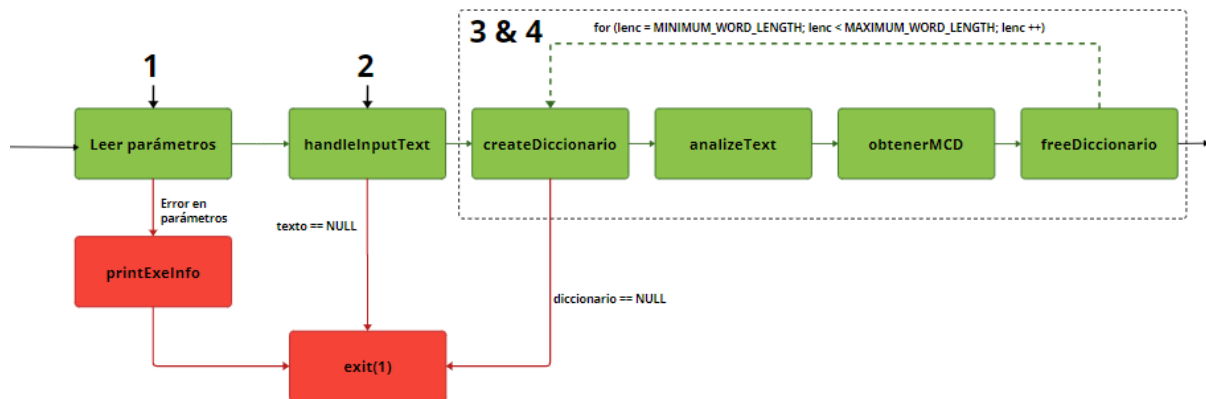
- AnalyzeText():

Esta función procesa el texto cifrado. Analiza todas las cadenas del texto con longitud igual a *lenc*, y las va guardando en un diccionario. En caso de encontrar una repetición, simplemente añade su posición al array de posiciones de esa cadena. En caso de no encontrar una repetición, añade la cadena al array de cadenas y guarda su posición.

- ObtenerMCD():

Obtiene el máximo común divisor de la distancia de todas las cadenas para las que haya al menos 3 repeticiones (para tener mínimo 2 distancias para calcular el MCD). En caso de que el MCD de las distancias sea igual a 1 (las distancias son coprimas entre sí), no se tiene en cuenta esa cadena.

## Implementación del algoritmo:



### 1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./criptoanálisisKasinski [-i filein] [-o fileout] \n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Como se puede ver, únicamente requiere 2 parámetros opcionales ("-i" y "-o"), por si se quiere asignar un fichero de entrada o salida.

### 2. Leer texto de entrada

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}
```

Empleamos la función '*handleInputText()*' mencionada previamente para obtener el texto cifrado por el método Vigenere que vamos a criptoanalizar.

3. Ejecutar el algoritmo iterando sobre los diferentes tamaños de palabra especificados por las macros 'MINIMUM\_WORD\_LENGTH' y 'MAXIMUM\_WORD\_LENGTH'.

```
for (int lenc = MINIMUM_WORD_LENGTH; lenc < MAXIMUM_WORD_LENGTH; lenc++) {
    diccionario = createDiccionario(lenc, strlen(texto));
    if (!diccionario) {
        free(texto);
        exit(1);
    }

    if (f) {
        fprintf(f, "Longitud de las palabras probado: %d\n", lenc);
        fprintf(f, "-----\n");
    } else {
        printf("Longitud de las palabras probado: %d\n", lenc);
        printf("-----\n");
    }

    analyzeText(texto, diccionario, lenc);

    obtenerMCD(diccionario, lenc, strlen(texto), f);

    freeDiccionario(diccionario, lenc, strlen(texto));

    if (f) {
        fprintf(f, "\n-----\n");
    } else {
        printf("\n-----\n");
    }
}
```

Como se puede observar, creamos un diccionario por cada iteración, analizando el texto y obteniendo el MCD de las distancias entre las diferentes palabras para cada tamaño de palabra especificado en cada iteración. De esta forma, somos capaces de obtener una aproximación del tamaño de clave usado para cifrar el texto.

4. Extraer los resultados en el fichero establecido

A lo largo del programa, hemos ido imprimiendo los resultados por la salida especificada al ejecutar el programa. Esta puede ser o bien la salida estándar si no se ha incluido el parámetro "-o", o el fichero especificado en ese parámetro.

```
if (f) {
    fprintf(f, "Longitud de las palabras probado: %d\n", lenc);
    fprintf(f, "-----\n");
} else {
    printf("Longitud de las palabras probado: %d\n", lenc);
    printf("-----\n");
}
```

Esto es un ejemplo de cómo se procesa la salida del programa. La variable f es el archivo de salida del texto en caso de haber especificado uno con el parámetro '-o'.



## Criptografía y resultados

Para probar este método, podemos hacer uso del makefile ('make run\_CriptografíaKasinski'). Por defecto, este comando utiliza el archivo de entrada 'adios.txt' (teniendo en cuenta que previo a esta llamada se ha realizado un cifrado de Vigenere y guardado en este archivo), y 'salida.txt' como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta 'textos' si se desea, o probar con textos propios.

Ahora vamos a probar la efectividad de este método bajo distintos entornos:

- Probamos el método con un texto largo ('hamlet.txt') y con una clave relativamente corta de 8 caracteres (abcdefgh).

```
Criptografía de cifrado Vigenere Kasinski
Longitud mínima de palabras: 3
Longitud máxima de palabras: 6
.....
Longitud de las palabras probado: 3
.....
MCD de la cadena ujh: 8 | Distancias 904 1560 4224 4288 4528 4736 4752 5488 5712 5768 6216 7928 8344 9336 9520 10360 10392 10944 10960 11248 11736 11912 11920 13384 14576 16
MCD de la cadena jhx: 4 | Distancias 24224 44024 51880 51928 52052 55288 73616 80656 81752 96832 110912 125320
MCD de la cadena gnm: 4 | Distancias 296 4388 7880 20324 53420 53888 65624 78432 87412 91568 100764 121116 122504 122908 125468 129632
MCD de la cadena nme: 8 | Distancias 24456 48384 65624 122504 129632
MCD de la cadena ear: 8 | Distancias 792 7144 10584 17320 18640 22776 60992 81408 83256 86616 100920
MCD de la cadena arj: 8 | Distancias 18344 18480 18600 60992 94344 109904 118912 124904
MCD de la cadena rjn: 8 | Distancias 1352 33640 63184 76336 93920 101176 103192 109880 109904 118872 118912
MCD de la cadena ttf: 2 | Distancias 9008 18432 24760 34584 38096 40784 42126 55400 63944 71534 72246 81230 89584 90360 99592 101424 104278 108200 119742 125776 126064 12693
MCD de la cadena vsv: 8 | Distancias 32664 38552
MCD de la cadena jmp: 2 | Distancias 1270 19630 25200 47662 57216 101134 109190 116232 119408
MCD de la cadena mph: 8 | Distancias 7912 9864 15920 16184 20856 25200 26536 31416 36888 57216 61104 66888 78120 82416 90640 105480 119408
MCD de la cadena iss: 8 | Distancias 40384 41512 45176 47088 75352 102544
.....
Longitud de las palabras probado: 4
.....
MCD de la cadena ujh: 8 | Distancias 24224 55288 73616 81752 96832 110912 125320
.....
Longitud de las palabras probado: 5
.....
Longitud de las palabras probado: 6
.....
```

Como podemos ver, con claves no demasiado largas, y textos suficientemente largos, este método de criptoanálisis es muy efectivo, devolviéndonos la mayoría de las veces un tamaño de clave correcto

- Probamos el método con un texto corto ('corto.txt') y con una clave de 5 caracteres (abcde).

```
Criptoanálisis de cifrado Vigenere Kasinski
Longitud mínimas de palabras: 2
Longitud máxima de palabras: 6
-----
Longitud de las palabras probado: 2
-----
-----
Longitud de las palabras probado: 3
-----
-----
Longitud de las palabras probado: 4
-----
-----
Longitud de las palabras probado: 5
-----
-----
Longitud de las palabras probado: 6
-----
-----
```

Cuando tenemos un texto corto este método pierde eficiencia, ya que es menos probable que 2 o más cadenas del texto iguales hayan sido cifradas con la misma parte de la clave. Esto se ve claramente en la imagen, ya que no hemos sido capaces de encontrar ni una repetición en el texto cifrado.

- Probamos el método con un texto y una clave largos, de 100 caracteres (abcdefghijklmnopqrstuvwxyzqwertyuiopasdfghjklzxcvbnmpoiuytrewqlkjhgfdsa mnbvcxzosefdbnlknbvodiapnspc).

```
Criptoanálisis de cifrado Vigenere Kasinski
Longitud mínimas de palabras: 3
Longitud máxima de palabras: 6
-----
Longitud de las palabras probado: 3
-----
-----
Longitud de las palabras probado: 4
-----
-----
MCD de la cadena letp: 6 | Distancias 6582 18432
-----
Longitud de las palabras probado: 5
-----
-----
Longitud de las palabras probado: 6
-----
-----
```

Como se puede observar, cuanto mayor sea la clave, menos fiable será este método de criptoanálisis, ya que encontraremos menos repeticiones de cadenas. En caso de que la llave sea mayor o igual que la longitud del texto, es imposible encontrar el tamaño de esta por este método de criptoanálisis. Esto se debe a que no se va a encontrar ninguna

parte del texto que haya sido cifrado por la misma parte de la clave, y por ende no vamos a poder calcular las distancias entre cadenas repetidas del texto.

## **c.2. Índice de Coincidencia**

### **Previo a implementación**

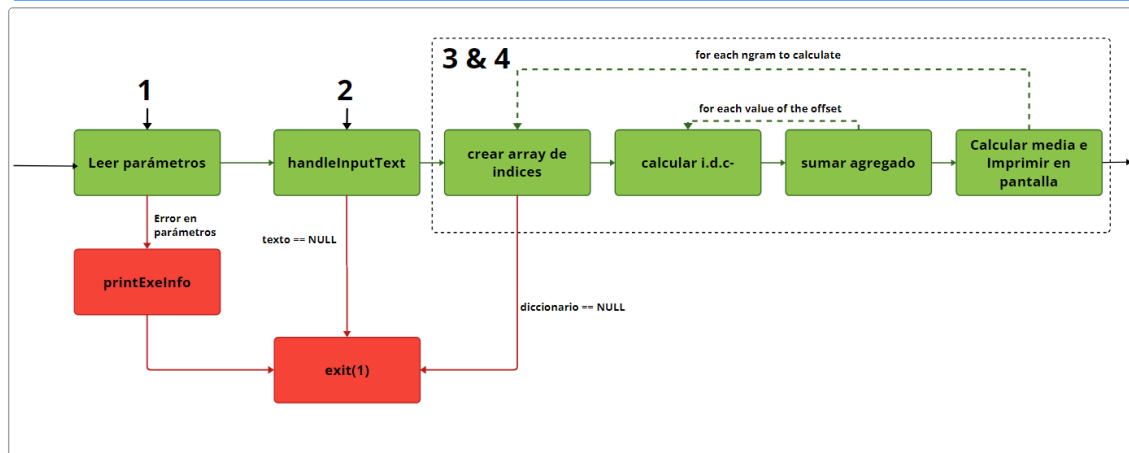
Calcular el índice de un texto será una tarea bastante simple y sencilla de programar. Para nuestro caso lo que haremos será crear una función que reciba un texto y un offset sobre el que empezar a calcular el índice de coincidencia (esto es porque suponemos que el texto está cifrado por Vigenere).

Algo importante a la hora de tener en cuenta con el índice de coincidencia (al igual que con el de Kasiski) es que si queremos obtener unos resultados satisfactorios necesitaremos un texto cifrado lo suficientemente grande. Para poder maximizar los recursos del texto que tengamos lo que haremos será calcular el índice de coincidencia para todos los offsets del bloque calculado. Es decir, si por ejemplo queremos verificar si el tamaño de bloque es 5, calcularemos los índices de coincidencia de los primeros caracteres, de los segundos etc y luego calcularemos la media aritmética para tener un resultado final. De esta forma, maximizaremos el tamaño del texto para tener una referencia lo más precisa posible, ya que cuanto mayor sean los bloques, menos texto y menos caracteres tendremos para calcular cada índice.

Para los ejemplos que mostraremos, hemos cifrado la obra “hamlet” al completo, por lo que el texto será lo suficientemente grande y representativo. En nuestro caso, como nuestros textos son siempre en inglés, esperaremos obtener un índice de coincidencia de 0,068 aproximadamente cuando hallemos el tamaño de bloque. Se podrá calcular tanto el índice de coincidencia de un tamaño de bloque o de todos los valores hasta dicho valor dependiendo de cómo se ejecute el programa.

## Implementación del algoritmo:

### CRIPTOANALISIS INDICES DE COINCIDENCIA



1. Primero nos aseguraremos de obtener los parámetros correctamente. Esta es la estructura que debe tener la ejecución del programa para que funcione.

```
void printExeInfo() {
    printf("./criptoanalisisIndices {-l Ngrama} {-m mod} [-i filein] ([-o fileout]) \n");
    printf("-l Ngrama: longitud del ngrama\n");
    printf("-m mod: modulo\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Se deben introducir el tamaño del bloque por el que se quiere buscar, el módulo del lenguaje que se está analizando, el fichero de entrada que se quiere criptoanalizar y la salida donde se imprimirán los resultados. Para el tamaño de bloque si se pone un número negativo, se criptoanalizarán todos los valores desde 1 hasta al valor señalado y el archivo de salida es opcional, si no se especifica uno se imprimirá por la salida estándar.

2. Leer texto de entrada

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}
```

Empleamos la función '*handleInputText()*' mencionada previamente para obtener el texto cifrado por el método Vigenere que vamos a criptoanalizar.

3. Ahora se calculará el índice de coincidencia para cada bloque. El índice de coincidencia es muy simple de calcular, únicamente guardaremos un array la frecuencia de cada carácter y luego los sumaremos y elevaremos al cuadrado siguiendo la fórmula.

La función devolverá el resultado que guardaremos para posteriormente calcular la media.

```
for(i=0;i<mod;i++) {
    valores[i] = 0;
}

for(i=offset; i<len; i+=n) {
    valores[texto[i] - 'a'] = valores[texto[i] - 'a'] + 1;
}

for(i=0;i<mod;i++) {
    res += (valores[i] / (len/n))*(valores[i] / (len/n));
}

free(valores);

return res;
```

4. Calcularemos la media de todos los índices de coincidencia e imprimiremos el resultado final. Y se volverá al paso 2 hasta que se hayan completado para todos los “ngramas”. El usuario deberá leer los resultados, interpretarlos y deducir cuál puede ser el tamaño del bloque cifrado (si es que es alguno).

```
/* Various ngramas*/
else {
    fprintf(out, "Criptonalisis de cifrado Vignere metodo Indice de Coincidencia\n");
    fprintf(out, "-----\n");
    for (i=1; i<=ngrama; i++){
        res = 0;
        for(int j=0; j<i; j++){
            res += calcular_indices(texto, i, j, modul);
        }
        fprintf(out, "El indice de coincidencia (media de todos los offset) para ngrama %d calculado es: %f\n", i, res/(i-1));
    }

    fprintf(out, "-----\n");
    fprintf(out, "Indice de coincidencia lenguaje ingles: 0.065\n");
    fprintf(out, "Indice de coincidencia lenguaje aleatorio: 0.038\n");
}
```

## Criptografía y resultados

Para probar este método, podemos hacer uso del makefile (‘make run\_analisis\_indices’) para facilitar su ejecución y se pueden cambiar los parámetros desde el fichero. Las ejecuciones mostradas en la memoria son de un texto muy largo (hamlet entero) por lo que el índice de coincidencia que adivina el tamaño de bloque es muy distinto al resto y cercano a 0.068.

```
nacho@nacho-HP-250-G7-Notebook-PC:~/Desktop/CRIPTO/v0.6/CRIPTO-PI$ make run_criptoanalisisIndices
vignere/criptoanalisisIndices -l -15 -m 26 -i textos/adios.txt
Criptonalisis de cifrado Vignere metodo Indice de Coincidencia
-----
El indice de coincidencia (media de todos los offset) para ngrama 1 calculado es: 0.042939
El indice de coincidencia (media de todos los offset) para ngrama 2 calculado es: 0.047338
El indice de coincidencia (media de todos los offset) para ngrama 3 calculado es: 0.042953
El indice de coincidencia (media de todos los offset) para ngrama 4 calculado es: 0.064066
El indice de coincidencia (media de todos los offset) para ngrama 5 calculado es: 0.042964
El indice de coincidencia (media de todos los offset) para ngrama 6 calculado es: 0.047368
El indice de coincidencia (media de todos los offset) para ngrama 7 calculado es: 0.042985
El indice de coincidencia (media de todos los offset) para ngrama 8 calculado es: 0.064107
El indice de coincidencia (media de todos los offset) para ngrama 9 calculado es: 0.043001
El indice de coincidencia (media de todos los offset) para ngrama 10 calculado es: 0.047396
El indice de coincidencia (media de todos los offset) para ngrama 11 calculado es: 0.043011
El indice de coincidencia (media de todos los offset) para ngrama 12 calculado es: 0.064131
El indice de coincidencia (media de todos los offset) para ngrama 13 calculado es: 0.043051
El indice de coincidencia (media de todos los offset) para ngrama 14 calculado es: 0.047430
El indice de coincidencia (media de todos los offset) para ngrama 15 calculado es: 0.043050
-----
Indice de coincidencia lenguaje ingles: 0.065
Indice de coincidencia lenguaje aleatorio: 0.038
```

Para este ejemplo se ha utilizado la obra entera de Hamlet que se ha cifrado por Vignere con bloque 4. Como podemos ver en la salida, el resultado del ngrama 4 (y sus múltiplos) se acercan mucho al índice de coincidencia inglés y el resto más cercanos a los aleatorios.

### 3. Cifrado de flujo

#### Previo a implementación

Este método necesita generar y manipular streams aleatorios para poder realizar un flujo de claves a la hora de cifrar y descifrar un texto. Es por ello por lo que hemos creado las siguientes funciones, los cuales simulan un stream LFSR:

- GenerateStream():

```
void generateStream(long seed, char *stream) {  
    srand(seed);  
    for (int i = 0; i < STREAM_LENGTH; i++) {  
        stream[i] = rand() % 26 + 'a';  
    }  
}
```

Este método genera un stream (array de chars), de tamaño fijado por la macro 'STREAM\_LENGTH'. Para la generación de parámetros aleatorios, la función recibe una semilla, la cual le pasamos al método 'srand()'. Gracias a esto, podemos usar la función 'rand()' para generar caracteres aleatorios, e inicializar los valores del stream.

- PrintStream():

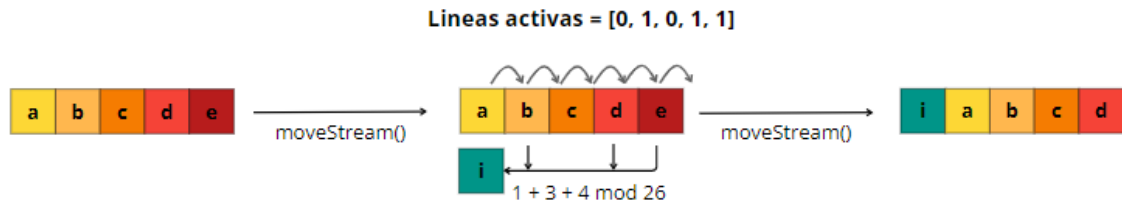
```
void printStream(char *stream) {  
    for (int i = 0; i < STREAM_LENGTH; i++) {  
        printf("%c", stream[i]);  
    }  
    printf("\n");  
}
```

Este método imprime todos los valores del stream. Principalmente este método se ha usado para tareas de debugging.

- MoveStream():

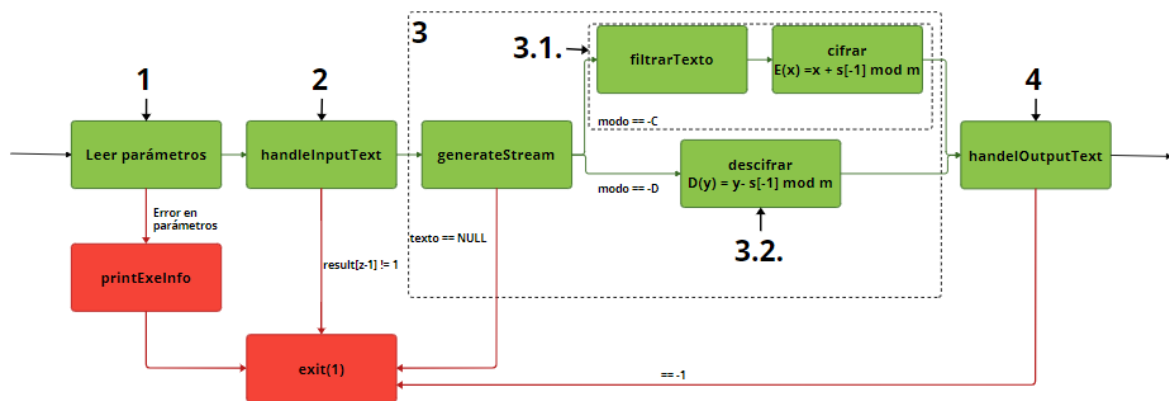
Este método se usa para generar nuevas claves. Mueve todos los elementos del stream una posición a la derecha. Para generar el primer elemento, hacemos uso de un array de ints('líneas-activas') que contiene 0s y 1s. Si una línea está activa (= 1), significa que se va a hacer uso del elemento que se encuentra en esa posición para generar el nuevo elemento. Se sigue la siguiente fórmula para la generación:

$$s[0] = \sum_{i=0}^{lin. \text{ activas}} s[i] \bmod m$$



## Implementación del algoritmo

Ahora, vamos a explicar paso a paso como hemos integrado este método, siguiendo el esquema:



1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./flujo {-C|-D} {-m [Zm]} {-s seed} {-l líneas activas} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m [Zm]: tamaño del espacio de texto cifrado\n");
    printf("-s seed: semilla para el generador de números pseudoaleatorios / de la cadena de flujo (cuanto mayor sea mejor)\n");
    printf("-l líneas activas: Líneas que van a estar activas. 1 significa activa, 0 inactiva (Ej. 1001101101 para tamaño de stream 10)\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Como se puede ver, contamos con 4 parámetros obligatorios (“-C|-D”, “-m”, “-s”, “-l”) y 2 opcionales (“-i”, “-o”).

2. El siguiente paso es obtener el texto que vamos a cifrar.

```
/* Leer texto de entrada */
texto = handleInputText(i);
if (!texto) {
    exit(1);
}
```

3. Procedemos a generar el stream y a cifrar o descifrar dependiendo del modo de ejecución introducido con el parámetro ‘-m’:



```

generateStream(seed, stream);

/* Cifro o descifro */
if (strcmp(modo, "-C") == 0) {

    filtrarTexto(texto);

    cifrar(m, stream, texto, lineas_activas);
} else {
    descifrar(m, stream, texto, lineas_activas);
}

```

### 3.1. Caso de cifrado:

Primero, debemos filtrar el texto de entrada con la función “*filtrarTexto*” mencionada previamente. Una vez realizado este paso, procedemos a cifrar el texto mediante la fórmula explicada en teoría, aplicándosela a cada carácter del texto.

$$E(x) = x + stream[-1] \bmod m$$

Tras cada iteración será necesario llamar a la función ‘*moverStream()*’ para generar nuevas claves.

### 3.2. Caso de descifrado:

En este caso no es necesario filtrar el texto, ya que en principio este paso ya se ha realizado a la hora de cifrarlo. Podemos proceder directamente a descifrar el texto usando la fórmula vista en clase.

$$D(y) = y - stream[-1] \bmod m$$

Al igual que a la hora de cifrar, será necesario llamar a la función ‘*moverStream()*’ tras cada iteración.

4. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

```

/* Escribir nuevo texto en salida*/
if (handleOutputText(o, texto) == -1) {
    exit(1);
}

```

## **Criptoanálisis y resultados**

Para probar este método, podemos hacer uso del makefile ('make run\_flujo\_C' para cifrar, 'make run\_flujo\_D' para descifrar). Por defecto, estos comandos utilizan el archivo de entrada 'hola.txt', y 'adios.txt' como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta 'textos' si se desea, o probar con textos propios.

Este método no deja de ser un cifrado de Cesar con la única modificación de que la clave cambia siguiendo un flujo.

La seguridad de este algoritmo depende principalmente de la longitud del stream y del número de líneas de activación escogidas. Esto se debe a que estos 2 parámetros marcan el periodo de la secuencia cifrante. Cuanto mayor sea esta, mayor fortaleza tendrá el cifrado siendo más resistente a criptoanálisis. Un periodo más largo significa que un atacante necesitaría más datos y más tiempo para detectar repeticiones y deducir la clave.

Otro de los factores que afecta a la fortaleza de este método es la generación del stream inicial. Cuanto mejor sea el método de generación de valores aleatorios usado para crear el stream más seguro será este cifrado.

## 4. Método de transposición

### Previo a implementación

Para implementar este método necesitamos una forma de generar permutaciones aleatorias de tamaño variable (en nuestro caso solo podemos usar permutaciones de tamaño 2 o 3). Además, al hacer uso del método de Hill para implementar este cifrado, necesitamos una forma de transformar esta permutación en una matriz que cumpla la misma función.

Hemos desarrollado los siguientes métodos para implementar este método:

- Random\_num():

```
int random_num(int inf, int sup)
{
    int res = inf;

    /*Error control*/
    if (sup < inf)
    {
        return -1;
    }

    res += rand() % (sup - inf + 1);
    return res;
}
```

Genera un número aleatorio dado un límite inferior ('inf') y un límite superior ('sup').

- GeneratePermutation():

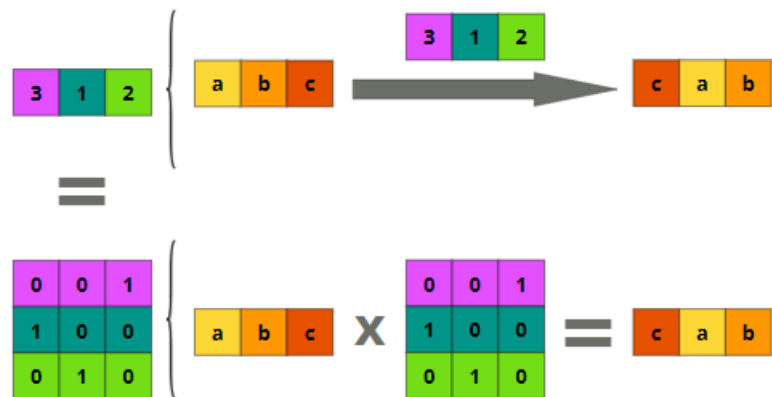
En caso de querer generar una permutación aleatoria en lugar de una creada por nosotros, podemos hacer uso de este método. Esta función hace uso de 'random\_num()' para generar una permutación aleatoria de tamaño 'n'.

- InvertPermutation():

Para poder descifrar un texto cifrado por una permutación dada, debemos ser capaces de crear una permutación inversa. Para eso se utiliza este método.

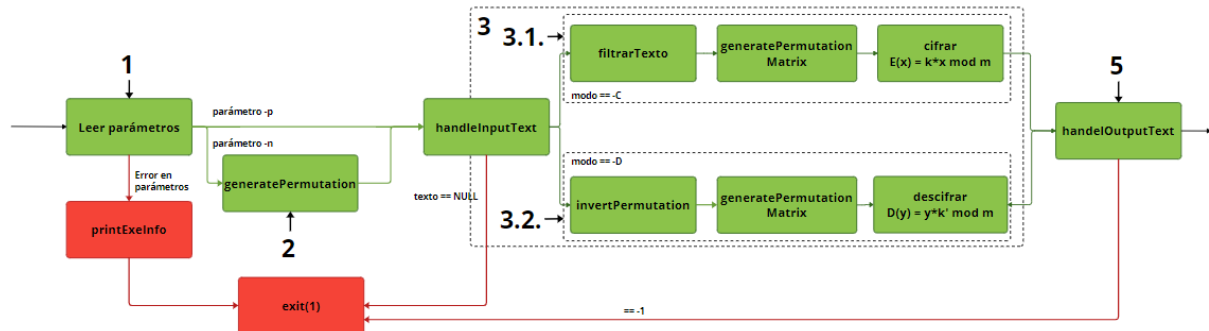
- `GeneratePermutationMatrix()`:

Como bien hemos mencionado antes, para implementar este método hemos hecho uso del cifrado de Hill. Por ello debemos de ser capaces de generar una matriz dada una permutación. Siguiendo la siguiente propiedad, podemos generar una matriz de permutación:



## Implementación del algoritmo

Ahora, vamos a explicar paso a paso como hemos integrado este método, siguiendo el esquema:



1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./transposicion {-C|-D} {-p permutación | -n Nperm}[-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-p permutación: permutación a aplicar (P.ej. 213)\n");
    printf("-n Nperm: tamaño de vector de permutaciones (entre 2 y 3)\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Como se puede ver, contamos con 2 parámetros obligatorios (“-C|-D”, “-p | -n”) y 2 opcionales (“-i”, “-o”).

2. El siguiente paso es obtener la permutación, para lo que hay 2 opciones. En caso de haber recibido el parámetro ‘-p’, se usa la permutación pasada como argumento. Por lo contrario, si hemos recibido el parámetro ‘-n’, hacemos uso de la función ‘generatePermutation()’, con tamaño igual al pasado como argumento.

```

if (strcmp(argv[2], "-p") == 0) {
    permutacion = argv[3];
    n = strlen(permutacion);
    if (n < 2 || n > 3) {
        printf("La permutación debe tener 2 o 3 elementos\n");
        exit(1);
    }

    permutacionint = (int *)malloc(n * sizeof(int));

    for (int x = 0; x < n; x++) {
        permutacionint[x] = permutacion[x] - '0';
    }

} else if (strcmp(argv[2], "-n") == 0) {
    n = atoi(argv[3]);
    if (n < 1) {
        printf("El número de permutaciones debe estar entre 2 y 3 (limitación de tamaño de matriz en hill\n");
        exit(1);
    }

    permutacionint = (int *)malloc(n * sizeof(int));
    if (permutacionint == NULL) {
        printf("Error en la asignación de memoria\n");
        exit(1);
    }

    generatePermutation(n, permutacionint);
}

```

3. Procedemos a obtener el texto pasado y a cifrar o descifrar dependiendo del modo de ejecución introducido con el parámetro ‘-m’:

### 3.1. Caso de cifrado:

Primero, debemos filtrar el texto de entrada con la función “*filtrarTexto*” mencionada previamente. Una vez realizado este paso, debemos añadir relleno al texto, por el mismo motivo que debíamos hacerlo para el método de Hill. Posteriormente, llamamos a la función ‘*generatePermutationMatrix()*’ para generar la matriz de permutación que vamos a usar para cifrar. Finalmente, procedemos a cifrar el texto mediante la fórmula explicada en teoría, aplicándosela a cada cadena del texto.

$$E(x) = x[1][i] \times k[i][j] \bmod m$$

### 3.2. Caso de descifrado:

En este caso no es necesario filtrar el texto, ya que en principio este paso ya se ha realizado a la hora de cifrarlo. El primero paso es obtener la matriz inversa a la que hemos usado para cifrar, para lo cual llamamos a la función ‘*invertPermutation()*’. Posteriormente, y al igual que a la hora de cifrar, llamamos a la función ‘*generatePermutationMatrix()*’ para generar la matriz de permutación. Finalmente, procedemos a cifrar el texto mediante la fórmula explicada en teoría, aplicándosela a cada cadena del texto:

$$D(y) = y[1][i] \times k^{-1}[i][j] \bmod m$$

4. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

```
/* Escribir nuevo texto en salida*/  
if (handleOutputText(o, texto) == -1) {  
    exit(1);  
}
```

## Criptanálisis y resultados

Para probar este método, podemos utilizar los siguientes comandos del makefile:

Para cifrar:

- `make run_transposicion_C_p`: Usa la permutación pasada como argumento.
- `make run_transposicion_C_n`: Genera una permutación aleatoria de tamaño `n`.

Para descifrar:

- `make run_transposicion_D_p`: Usa la permutación pasada como argumento.

Por defecto, estos comandos utilizan el archivo de entrada 'hola.txt', y 'adios.txt' como salida. Ambos se pueden cambiar por cualquier archivo que se encuentre dentro de la carpeta 'textos' si se desea, o probar con textos propios.

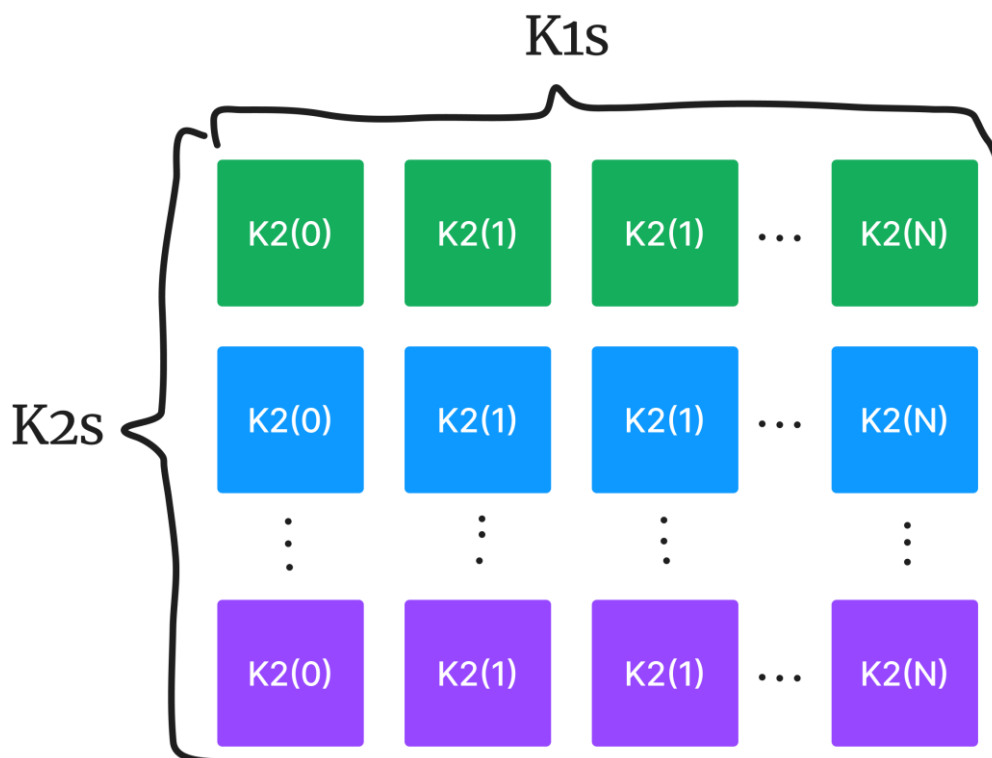
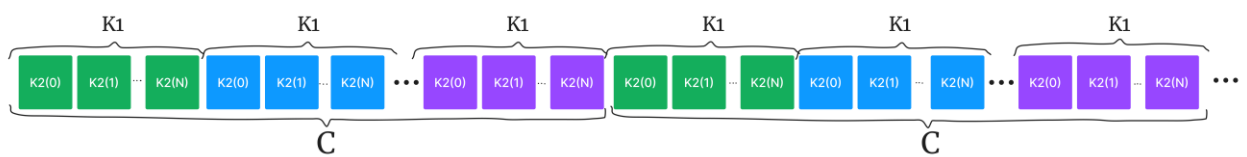
Además, no existe una versión de descifrado que genera permutaciones aleatorias. Esto es ya que no tendría demasiado sentido usar una permutación aleatoria para descifrar un texto que ha sido cifrado con una permutación en concreto.



## 5. Producto de criptosistemas permutación

### Previo a implementación

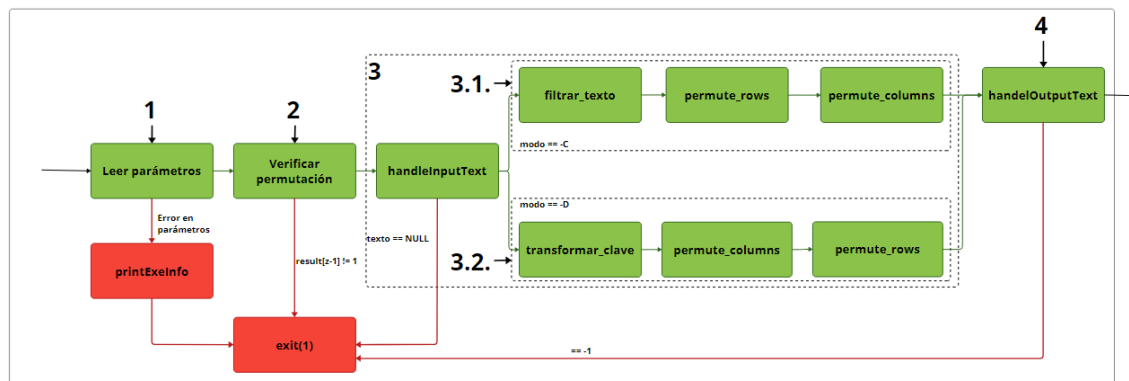
A continuación, vamos a implementar un algoritmo de cifrado por permutación. Como indica el enunciado, vamos a utilizar un método doble de permutaciones, primero permutando las filas del texto y posteriormente las columnas. Para plantear este algoritmo debemos imaginarnos que el texto está fragmentado en bloques a los que llamaremos 'C'. Cada bloque 'C', a su vez, está subdividido en bloques 'K1' a los que denominaremos 'filas' durante la explicación. Por último, cada elemento (carácter) que contiene cada K1 será el elemento de una columna. Es decir, el primer elemento de todos los K1 de una 'C' forman el primer bloque K2 de la matriz.



*Esta última imagen sería una única C*

Los esquemas anteriores ejemplifican la estructura de K1 y K2 para comprender el método. Se debe entender que cada casilla es un carácter del texto que tiene tantas Cs como sea necesario.

## Implementación del algoritmo



### 1. Parámetros necesarios para la ejecución del algoritmo:

```
void printExeInfo() {
    printf("./permutacion [-C|-D] {-m |Zm|} {-k filek} [-i filein] [-o fileout]\n");
    printf("-C|-D: -C para cifrar, -D para descifrar\n");
    printf("-m |Zm|: tamaño del espacio de texto cifrado\n");
    printf("-k filek: archivo donde esta guardado la matriz para cifrar o descifrar el texto.\nSi se desea crear unas claves aleatorias introduzcase dos numeros separados por un espacio\n");
    printf("-i filein: archivo de entrada\n");
    printf("-o fileout: archivo de salida\n");
}
```

Para la ejecución de este método será necesario indicar si se quiere cifrar o descifrar, el módulo del lenguaje que se esté cifrando y los archivos de entrada y salida como en ejercicios anteriores. Este contiene la particularidad de que se debe indicar la clave que debe ser el nombre del archivo que la contiene. Las claves tienen que ser dos permutaciones en diferentes líneas separados por espacios y que empiecen por 0 y acaben por size-1. Si se indica un nombre que no sea un archivo, se calculará unas permutaciones al azar y se mostrarán por pantalla.

2. Luego, verificaremos que las claves son correctas, es decir que ambas se traten de permutaciones de cualquier tamaño pero que empiecen por 0 y acaben por el 'tamaño-1'. En caso contrario lo informaremos por pantalla y finalizará el programa.

3. Procedemos a leer el texto de entrada y desciframos o ciframos el texto en función del parámetro introducido:

Tanto cifrar como descifrar es sencillo de entender, aunque puede resultar algo confuso de comprender al programarlo.

#### 3.1 Caso de cifrado:

Lo primero de todo que tendremos que hacer será filtrar el texto y asegurarnos de que contiene un tamaño múltiplo de  $size\_k1 * size\_k2$  para tener suficientes caracteres para hacer las permutaciones. Se añadirán el número de caracteres necesarios hasta llegar dicho valor. Luego, utilizaremos la clave k1 para permutar entre los bloques interiores

de C, y una vez los hayamos terminado de intercambiar permutaremos cada bloque siguiendo la permutación de k2. El código parece confuso, pero se ha intentado hacer lo más claro posible con comentarios y realizando los cálculos uno a uno. Para la primera permutación usamos la función ‘permute\_rows’ que se encarga de permutar los bloques K1 entre sí, y para la segunda ‘permute\_columns’ que permuta cada bloque K1 interiormente.

```
void cifrar(char *texto, int mod, int *k1, int size_k1, int *k2, int size_k2) {
    int len = strlen(texto);
    int i;

    char *cifrado = (char *)calloc(len, sizeof(char));
    if (cifrado == NULL) {
        return;
    }

    permute_rows(texto, cifrado, k1, size_k1, size_k2);

    /* Save matrix to be able to swap*/
    for (i = 0; i < len; i++) {
        texto[i] = cifrado[i];
    }

    permute_columns(texto, cifrado, k2, size_k1, size_k2);

    /*strcpy(texto, cifrado)*/
    for (i = 0; i < len; i++) {
        texto[i] = cifrado[i];
    }

    free(cifrado);
}

void permute_columns(char *texto, char* descifrado, int *k, int size_k1, int size_k2){
    int i, aux, aux2;
    int len = strlen(texto);

    /* Permute columns */
    for (i = 0; i < len; i++) {
        aux = (i / size_k2) * size_k2;
        aux2 = i % size_k2;

        descifrado[i] = texto[aux + k[aux2]];
    }
}

void permute_rows(char *texto, char* cifrado, int *k, int size_k1, int size_k2) {
    int i, aux, aux2, aux3;
    int len = strlen(texto);

    /* Permute rows */
    for (i = 0; i < len; i++) {
        /*Calculate the block where we are*/
        aux = i/(size_k1*size_k2);
        aux *= size_k1*size_k2;

        /*Move to position*/
        aux2 = (i / size_k2) % size_k1;
        aux3 = i % size_k2;

        cifrado[i] = texto[aux + k[aux2]*size_k2 + aux3];
    }
}
```

### 3.2 Caso de descifrado:

El caso de descifrado es idéntico al de cifrado, pero se realizan las permutaciones de manera inversa. Es decir, primero reordenaremos las columnas y luego las filas. Sin embargo, tendremos que realizar un paso previo que será transformar las claves en su clave inversa. Esto nos permitirá volver a recolocar los caracteres y descifrar satisfactoriamente el texto.

```

void descifrar(char *texto, int mod, int *k1, int size_k1, int *k2, int size_k2) {
    int len = strlen(texto);
    int i;

    char *descifrado = (char *)calloc(len, sizeof(char));
    if (descifrado == NULL) {
        return;
    }

    invertPermutation(k1, size_k1);
    invertPermutation(k2, size_k2);

    /* Permute columns */
    permute_columns(texto, descifrado, k2, size_k1, size_k2);

    /* Save matrix to be able to swap*/
    for (i = 0; i < len; i++) {
        texto[i] = descifrado[i];
    }

    /* Permute rows */
    permute_rows(texto, descifrado, k1, size_k1, size_k2);

    /*strcpy(texto, descifrado)*/
    for (i = 0; i < len; i++) {
        texto[i] = descifrado[i];
    }

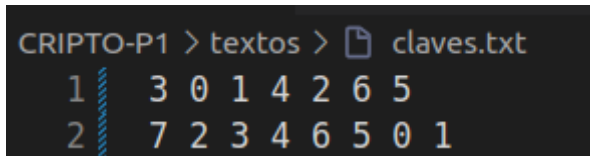
    free(descifrado);
}

```

4. Por último, quedaría devolver este texto tras haber sido transformado. Esto lo haremos mediante la función “*handleOutputText*”, mencionada previamente.

## Criptanálisis y resultados

Para probar este método se pueden utilizar los comandos del makefile “run\_permutacion\_C” y “run\_permutacion\_D” y se deben especificar las claves en “claves.txt” correctamente formateado, siendo la primera clave la primera línea y la segunda la última. Tener cuidado con los espacios de más, tabulaciones etc. Como por ejemplo:



```

CRIPTO-P1 > textos > claves.txt
1 3 0 1 4 2 6 5
2 7 2 3 4 6 5 0 1

```

Con esto podremos cifrar y descifrar los textos. Como dato adicional, que podemos comentar, la función “addFilling()” que añade el número de caracteres necesarios para cuadrar las cuentas siempre añade una ‘x’. Sería más interesante que añadiera caracteres al azar, ya que la posición de las ‘x’s al final del documento podría dar muchas pistas de cuáles son las claves del cifrado.