1. Contexto general: ¿Qué hace este programa?

Este código en JavaScript implementa una **demostración visual e interactiva** de cifrado clásico. Permite al usuario:

- Escribir un texto libre.
- Escoger el método de cifrado (César o Transposición por columnas).
- Cifrar o descifrar el mensaje.
- Ver paso a paso cómo se transforma el texto original en el cifrado.

El objetivo didáctico es mostrar cómo cada algoritmo transforma los datos **sin depender de bibliotecas externas**, solo usando lógica pura.

2. Cifrado César (Sustitución simple)

Concepto teórico

El **Cifrado César** es un método de **sustitución monoalfabética**. Consiste en **desplazar cada letra** del texto un número fijo de posiciones en el alfabeto.

Por ejemplo, con un desplazamiento de 3:

> Texto: `HOLA` → Cifrado: `KROD` H→K, O→R, L→O, A→D

Para descifrar, se aplica el desplazamiento inverso (-3).

Funcionamiento en el código

```
// 1) César: cifrar/descifrar (solo letras latinas A-Z / a-z; otros caracteres se
mantienen)
function caesarEncryptPure(text, shift){
  // Aseguramos shift entre 0-25
  const s = ((Number(shift) % 26) + 26) % 26;
  let out = '';
  for (let ch of text){
    const code = ch.charCodeAt(0);
    if (code >= 65 && code <= 90){
      // A-Z
      out += String.fromCharCode((code - 65 + s) % 26 + 65);
    } else if (code >= 97 && code <= 122){
      // a-z
      out += String.fromCharCode((code - 97 + s) % 26 + 97);
    } else {
      out += ch; // números, espacios y símbolos se mantienen
    }
  }
  return out;
}
```

Explicación línea por línea

Desglose detallado:

- `function caesarEncryptPure(text, shift){`
  - Declara una función llamada `caesarEncryptPure` que recibe:
    - `text`: el texto a cifrar (string).
    - `shift`: el desplazamiento (número de posiciones).
- `const s = ((Number(shift) % 26) + 26) % 26;`
  - `Number(shift)`: convierte `shift` a número (por si viene como string desde un input).
  - `% 26` asegura que un desplazamiento grande vuelva a mapearse dentro de 0..25 (porque el alfabeto tiene 26 letras).
  - `+ 26` y el segundo `% 26` sirven para manejar desplazamientos negativos correctamente.
    - Ejemplo: si `shift = -3`, `Number(shift) % 26` da `-3`. `(-3 + 26) % 26` → `23`. Así -3 se convierte en 23 (equivalente).
  - `s` es el desplazamiento final en el rango `0..25`.
- `let out = '';`
  - Inicializamos `out` como string vacío; en él iremos concatenando el texto cifrado.
- `for (let ch of text){`
  - Bucle que recorre carácter por carácter del string `text`.
  - `ch` es una letra, número o símbolo en cada iteración.
- `const code = ch.charCodeAt(0);`
  - `charCodeAt(0)` devuelve el código Unicode (prácticamente ASCII para letras latinas simples).
  - Ejemplo: `'A'.charCodeAt(0)` → `65`, `'a'.charCodeAt(0)` → `97`.
- `if (code >= 65 && code <= 90){`
  - Comprueba si `ch` es una letra mayúscula (`A`..`Z`).
  - 65..90 son los códigos ASCII para `A`..`Z`.
- `out += String.fromCharCode((code - 65 + s) % 26 + 65);`
  - Paso a paso:
    - `code - 65`: convierte `A..Z` a `0..25`. Ej: `A`→0, `B`→1.
    - `+ s`: aplica el desplazamiento.
    - `% 26`: asegura que si supera `25` se regrese al inicio del alfabeto (rotación).
    - `+ 65`: vuelve a convertir `0..25` a códigos ASCII `65..90`.
    - `String.fromCharCode(...)` convierte el código de vuelta a carácter.

    - Ejemplo concreto: `ch = 'Y'` (code=89), `s = 3`:
        - `(89 - 65 + 3) % 26 + 65` → `(24 + 3) % 26 + 65` → `27 % 26 + 65` → `1 + 65` → `66` → `'B'`.

- `} else if (code >= 97 && code <= 122){`
    - Caso para letras minúsculas (`a`..`z` tienen códigos 97..122).
- `out += String.fromCharCode((code - 97 + s) % 26 + 97);`
    - misma lógica que en mayúsculas pero usando base `97` para `a`.
- `} else { out += ch; }`
    - Si no es letra (puede ser número, espacio, signo), se deja tal cual y se concatena al resultado.
    - Esto preserva la legibilidad y el formato del texto (p. ej. puntuación y espacios).
- `return out;`
    - Devuelve el texto cifrado completo.

Descifrado

```
function caesarDecryptPure(text, shift){
  return caesarEncryptPure(text, -shift);
}
```

- `caesarDecryptPure` reutiliza el cifrador: descifrar con `shift` es equivalente a cifrar con `-shift`.
- Esto evita duplicar la lógica.

3. Transposición por Columnas

Concepto teórico

Este es un **método de cifrado por permutación**. No altera las letras, sino **su posición** en el mensaje.

Procedimiento:

1. Se escribe el texto en filas con un número fijo de columnas (clave).
2. Se lee por columnas para formar el mensaje cifrado.

Ejemplo (clave = 4):

T	E	X	T
O	S	E	C
I	F	R	A
D	O	X	X

Lectura por columnas → **TOID ESFO XERX CAXX** → **T O I D E S F O X E R X C A X X**

Cifrado = `TOIDESFOXERXCAXX`

Implementación del cifrado

```
function columnarEncryptPure(text, key){
  const k = Math.max(1, Math.floor(Number(key) || 1));
  let s = text;
  while (s.length % k !== 0) s += 'X';
  let out = '';
  for (let col = 0; col < k; col++){
    for (let i = col; i < s.length; i += k){
      out += s[i];
    }
  }
  return out;
}
```

Explicación

Desglose:

- `function columnarEncryptPure(text, key){`
  - `text`: mensaje original.
  - `key`: número de columnas (clave numérica).
- `const k = Math.max(1, Math.floor(Number(key) || 1));`
  - `Number(key)`: convierte `key` a número.
  - `|| 1`: si `key` es `0`, `null` o `''`, usa `1` por defecto.
  - `Math.floor(...)`: si el usuario pone decimal, lo redondea hacia abajo a entero.
  - `Math.max(1, ...)`: asegura que `k` al menos 1 (no puede ser 0 columnas).
  - Resultado: `k` es un entero ≥ 1 representando columnas.
- `let s = text;`
  - `s` es la copia del texto que se procesará y rellenará si hace falta.
- `while (s.length % k !== 0) s += 'X';`
  - Si la longitud de `s` no es divisible por `k`, se añaden caracteres `'X'` hasta que sí lo sea.
  - Propósito: completar la última fila para que la matriz quede rectangular.
  - Ejemplo: `text = "HOLA"` (4 letras), `k = 3` → 4 % 3 = 1 -> agrega 2 `X` → `HOLAXX` (6 caracteres, 2 filas de 3).
- `let out = '';`
  - Inicializa el string de salida.

- `for (let col = 0; col < k; col++){`
  - Recorre cada columna (0 a `k-1`).
- `for (let i = col; i < s.length; i += k){`
  - Recorre la cadena `s` empezando en el índice `col`, saltando de `k` en `k`.
  - Esto emula leer la matriz por columnas.
  - Ejemplo: `s = 'HOLAXX'`, `k=3`:
    - `col=0`: i=0,3 → s[0]='H', s[3]='A' → lee "HA"
    - `col=1`: i=1,4 → s[1]='O', s[4]='X' → "OX"
    - `col=2`: i=2,5 → s[2]='L', s[5]='X' → "LX"
  - Concatenando por columnas `out = 'HA' + 'OX' + 'LX' = 'HAOXLX'`.
- `return out;`
  - Devuelve el mensaje cifrado por lectura columnar.

Descifrado

```
function columnarDecryptPure(cipher, key){
  const k = Math.max(1, Math.floor(Number(key) || 1));
  const rows = Math.ceil(cipher.length / k);
  let parts = [];
  let i = 0;
  for (let c = 0; c < k; c++){
    parts.push(cipher.slice(i, i + rows));
    i += rows;
  }
  let out = '';
  for (let r = 0; r < rows; r++){
    for (let c = 0; c < k; c++){
      const piece = parts[c];
      if (r < piece.length) out += piece[r];
    }
  }
  return out.replace(/X+$/g, '');
}
```

◇ Explicación paso a paso

Desglose:

- `const rows = Math.ceil(cipher.length / k);`
  - Calcula el número de filas que tenía la matriz original.
  - Ejemplo: `cipher.length=6`, `k=3` → `rows = Math.ceil(6/3) = 2`.

- Comentario importante: en implementaciones más complejas, si no se rellenó la última fila con la misma cantidad, algunas columnas serían una letra más cortas. En tu cifrado (donde se añadió `X`) todas las columnas quedan con `rows` caracteres. Por eso `numPerCol = rows`.
- `let parts = []; let i = 0; for (let c = 0; c < cols; c++){ parts.push(cipher.slice(i, i + numPerCol)); i += numPerCol; }`
  - Divide el texto cifrado en `cols` segmentos consecutivos, cada uno representando una columna.
  - `slice(i, i + numPerCol)` toma la porción de la cadena.
  - Ejemplo: `cipher='HAOXLX'`, `k=3`, `rows=2`:
    - c=0 → slice(0,2) → 'HA'
    - c=1 → slice(2,4) → 'OX'
    - c=2 → slice(4,6) → 'LX'
  - `parts = ['HA','OX','LX']`.
- `let out = ''; for (let r = 0; r < rows; r++){ for (let c = 0; c < cols; c++){ const piece = parts[c]; if (r < piece.length) out += piece[r]; } }`
  - Reconstruye la matriz leyendo fila por fila:
    - Primero fila `r=0`: toma `parts[0][0]`, `parts[1][0]`, `parts[2][0]` → 'H','O','L'
    - Segunda fila `r=1`: `parts[0][1]`, `parts[1][1]`, `parts[2][1]` → 'A','X','X'
  - Resultado `out = 'HOLAXX'`.
- `return out.replace(/X+$/g, '');`
  - Elimina una o más `X` al final (`$` = fin de cadena).
  - `replace(/X+$/g, '')` borra esa secuencia de relleno.
  - De `'HOLAXX'` devuelve `'HOLA'`.

Casos límite, errores comunes y notas para defender en el aula

1. **Por qué normalizar el shift**: sin `((...)+26)%26` los desplazamientos negativos o mayores de 26 darían resultados inesperados. Explica con ejemplos numéricos.
2. **Preservar símbolos y espacios**: se diseñó para mantener la legibilidad y permitir combinar cifrados sin perder formato. También permite cifrar frases completas.
3. **Relleno con `X`**: la elección de `'X'` es arbitraria; podrías usar `'_'` u otro char que no aparezca normalmente. En la defensa, explica que el relleno evita matrices irregulares.
4. **Seguridad**:
   - César: vulnerable a fuerza bruta (solo 26 claves) y análisis de frecuencia.

- Columnar: más seguridad relativa (permuta) pero aún romántico/clásico; combínalo (p. ej. César luego Columnar) para explicar cómo se aumentaría la complejidad.

5. **Complejidad temporal**:
   - César: O(n) — recorre cada carácter una vez.
   - Columnar: O(n) también — rellena y recorre con saltos, pero cada carácter se procesa una cantidad constante de veces.

6. **Manejo de entradas inválidas**:
   - `key` y `shift` se convierten a números; si vienen vacíos se usan valores por defecto (evita `NaN`).

7. **Por qué usar `Array.from(document.querySelectorAll(...))`**:
   - para poder usar funciones de arrays (`map`, `filter`) que `NodeList` no siempre soporta en todos los navegadores.