CNS Assigments

Sure! I'll organize this as notes for you. We'll start with the first assignment, which is the **Caesar Cipher**. After going through this, you can ask for the next one.

1. W.A.P. to Implement Caesar Cipher

Concept:

The Caesar Cipher is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is 'shifted' by a certain number of positions down the alphabet.

Algorithm:

1. Choose a shift value (e.g., 3).
2. For each letter in the plaintext:
    - If it's an uppercase letter, shift it within the range of 'A' to 'Z'.
    - If it's a lowercase letter, shift it within the range of 'a' to 'z'.
3. Non-alphabetic characters remain unchanged.
4. The shifted letters form the ciphertext.

Example:

- **Plaintext:** HELLO
- **Shift:** 3
- **Ciphertext:** KHOOR

C++ Code Implementation:

```
#include <iostream>
using namespace std;

string caesarCipher(string text, int shift) {
    string result = "";
    for (int i = 0; i < text.length(); i++) {
        if (isupper(text[i])) {
            result += char(int(text[i] + shift - 65) % 26 +
65);
        } else if (islower(text[i])) {
            result += char(int(text[i] + shift - 97) % 26 +
97);
        } else {
            result += text[i];  // Non-alphabetic character
s remain unchanged
        }
    }
    return result;
}

int main() {
    string text = "HELLO";
    int shift = 3;
    cout << "Ciphertext: " << caesarCipher(text, shift);
    return 0;
}
```

Explanation:

- **isupper(text[i])** checks if the character is uppercase.
- **islower(text[i])** checks if the character is lowercase.
- **char(int(text[i] + shift - 65) % 26 + 65)** shifts the character for uppercase letters.
- **char(int(text[i] + shift - 97) % 26 + 97)** shifts the character for lowercase letters.
- The result is printed as the ciphertext.

---

2. W.A.P. to Implement Playfair Cipher with Key 'LDRP'

Concept:

The Playfair Cipher is a digraph substitution cipher, meaning it encrypts pairs of letters (digraphs) instead of single letters. This makes it more secure than simple substitution ciphers. The cipher uses a 5×5 grid of letters built using a keyword (in this case, 'LDRP'). Typically, the letters 'I' and 'J' are combined.

Algorithm:

1. **Create a 5×5 grid using the keyword:**
    - Start with the keyword (without duplicates).
    - Fill the rest of the grid with the remaining letters of the alphabet (combining 'I' and 'J').
2. **Prepare the plaintext:**
    - Split the plaintext into digraphs (pairs of letters).
    - If a pair has the same letter, replace the second one with 'X'.
    - If the plaintext has an odd number of characters, add an 'X' at the end.
3. **Encrypt the digraphs:**
    - **Rule 1:** If both letters are in the same row, replace them with the letters to their immediate right (wrap around if needed).
    - **Rule 2:** If both letters are in the same column, replace them with the letters immediately below them (wrap around if needed).
    - **Rule 3:** If the letters form a rectangle, replace them with the letters on the same row but at the opposite corners.

Example:

- **Plaintext:** HELLO
- **Key:** LDRP
- **Grid:**

```
L D R P A
B C E F G
H I K M N
O Q S T U
V W X Y Z
```

- **Ciphertext:** KEPMOF

C++ Code Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to create the Playfair table with the given key
void createPlayfairTable(char table[5][5], string key) {
    bool exists[26] = {false};
    exists['J' - 'A'] = true; // Usually 'I' and 'J' are co
mbined

    int idx = 0;
    for (char c : key) {
        if (!exists[c - 'A']) {
            table[idx / 5][idx % 5] = c;
            exists[c - 'A'] = true;
            idx++;
        }
    }

    for (char c = 'A'; c <= 'Z'; c++) {
        if (!exists[c - 'A']) {
            table[idx / 5][idx % 5] = c;
            idx++;
        }
    }
}

// Function to encrypt the plaintext using Playfair cipher
string playfairEncrypt(string text, char table[5][5]) {
    string result = "";

    // Pad 'X' if needed and split into digraphs
    if (text.length() % 2 != 0)
        text += 'X';
```

}

```
    for (int i = 0; i < text.length(); i += 2) {
        char a = text[i];
        char b = text[i + 1];
        if (a == b) b = 'X'; // Handling repeating characte
rs in a digraph

        int r1, c1, r2, c2;
        for (int r = 0; r < 5; r++) {
            for (int c = 0; c < 5; c++) {
                if (table[r][c] == a) {
                    r1 = r;
                    c1 = c;
                }
                if (table[r][c] == b) {
                    r2 = r;
                    c2 = c;
                }
            }
        }

        // Encrypt according to the Playfair rules
        if (r1 == r2) {
            result += table[r1][(c1 + 1) % 5];
            result += table[r2][(c2 + 1) % 5];
        } else if (c1 == c2) {
            result += table[(r1 + 1) % 5][c1];
            result += table[(r2 + 1) % 5][c2];
        } else {
            result += table[r1][c2];
            result += table[r2][c1];
        }
    }

    return result;
}

int main() {
    string key = "LDRP";  // Key for Playfair cipher
    char table[5][5];
    createPlayfairTable(table, key);

    string text = "HELLO";  // Plaintext
    cout << "Ciphertext: " << playfairEncrypt(text, table);
    return 0;
}
```

Explanation:

1. **createPlayfairTable:** This function generates the 5×5 grid using the key 'LDRP'. It fills in the grid, ensuring no letter is repeated (and combines 'I' and 'J').
2. **playfairEncrypt:** This function splits the plaintext into pairs (digraphs), applies the Playfair cipher rules, and generates the ciphertext.
3. **Handling Repeats:** If a digraph has repeating characters (like "LL" in "HELLO"), it replaces the second one with 'X'.
4. **Ciphertext:** The encrypted text is outputted at the end.

---

3. W.A.P. to Implement Polyalphabetic Cipher

Concept:

The Polyalphabetic Cipher is a type of substitution cipher that uses multiple alphabets for encryption. Unlike the Caesar Cipher, which shifts all characters by the same amount, the Polyalphabetic Cipher changes the shift value throughout the message. The most well-known form of this cipher is the **Vigenère Cipher**, where the shift for each letter depends on a repeating keyword.

Algorithm:

1. **Choose a keyword** that will determine the shifts for each letter in the plaintext.
2. **Repeat the keyword** until it matches the length of the plaintext.

3. For each letter in the plaintext:
    - Determine the shift value based on the corresponding letter of the keyword.
    - Apply the shift to the plaintext letter (similar to Caesar Cipher).
4. Combine the shifted letters to form the ciphertext.

Example:

- **Plaintext:** ATTACKATDAWN
- **Keyword:** LEMON
- **Extended Keyword:** LEMONLEMONLE
- **Ciphertext:** LXFOPVEFRNHR

C++ Code Implementation:

```
#include <iostream>
using namespace std;

// Function to generate the key in a repeated format to mat
ch the length of the plaintext
string generateKey(string text, string key) {
    int textLen = text.length();
    int keyLen = key.length();

    // Repeat the key to match the length of the text
    for (int i = 0; i < textLen - keyLen; i++) {
        key += key[i % keyLen];
    }
    return key;
}

// Function to encrypt the plaintext using Polyalphabetic
(Vigenère) cipher
string polyalphabeticCipher(string text, string key) {
    string result = "";
    for (int i = 0; i < text.length(); i++) {
        // Shift based on the corresponding letter of the k
ey
        char shift = (key[i] - 'A');  // Get shift value ba
sed on the key
        if (isupper(text[i])) {
            result += char((text[i] + shift - 'A') % 26 +
'A');
        } else if (islower(text[i])) {
            result += char((text[i] + shift - 'a') % 26 +
'a');
        } else {
            result += text[i];  // Non-alphabetic character
s remain unchanged
        }
    }
    return result;
}

int main() {
    string text = "ATTACKATDAWN";  // Plaintext
    string key = "LEMON";  // Keyword

    string extendedKey = generateKey(text, key);
    cout << "Ciphertext: " << polyalphabeticCipher(text, ex
tendedKey);
    return 0;
}
```

Explanation:

1. **generateKey:** This function extends the keyword to match the length of the plaintext. For example, if the plaintext is 12 characters long and the keyword is 5 characters long, the keyword is repeated as "LEMONLEMONLE".
2. **polyalphabeticCipher:** This function performs the encryption by shifting each character in the plaintext based on the corresponding character in the extended keyword. It uses the ASCII value of 'A' or 'a' to calculate the shift.
3. **Ciphertext:** The final encrypted message is printed.

Note:

- This implementation assumes the text is in uppercase for simplicity. If needed, you can modify the code to handle lowercase letters or non-alphabetic characters.

---

4. W.A.P. to Implement Hill Cipher (with Matrix Inversion)

Concept:

The Hill Cipher is a polygraphic substitution cipher that uses linear algebra to encrypt blocks of text. The encryption process involves matrix multiplication using a key matrix. Decryption requires finding the inverse of the key matrix.

Algorithm:

1. **Choose a Key Matrix:** The key matrix should be square (e.g., 2×2 or 3×3) and its determinant should be non-zero (i.e., it must be invertible modulo 26).
2. **Prepare the Plaintext:**
    - Convert the plaintext into vectors of numbers (e.g., 'A' = 0, 'B' = 1, ..., 'Z' = 25).
    - Split the plaintext into blocks that match the dimensions of the key matrix.
3. **Encrypt the Plaintext:**
    - Multiply each block vector by the key matrix (modulo 26) to get the ciphertext.
4. **Decrypt the Ciphertext:**
    - Find the inverse of the key matrix (modulo 26).
    - Multiply the ciphertext blocks by the inverse matrix to get the plaintext.

Example:

- **Key Matrix:**

```
| 6 24 1 |
| 13 16 10|
| 20 17 15|
```

- **Plaintext:** ACT
- **Ciphertext:** Use matrix multiplication to find the encrypted result.

C++ Code Implementation:

```
#include <iostream>
#include <vector>
#include <cassert>
using namespace std;

const int SIZE = 3;  // Size of the key matrix

// Function to find the modular inverse of a matrix
vector<vector<int>> matrixInverse(const vector<vector<int>>
& matrix, int mod) {
    // Helper functions for matrix inversion will be includ
ed here.
    // Finding the inverse is a complex process involving d
eterminants and adjugates.
    // For simplicity, assume you have a precomputed invers
e matrix or use a library.
}

// Function to encrypt plaintext using Hill cipher
string hillCipher(const string& text, const vector<vector<i
nt>>& keyMatrix) {
    vector<int> vec(SIZE);
    string result = "";

    // Convert plaintext to numerical values
    for (int i = 0; i < SIZE; i++) {
        vec[i] = text[i] - 'A';
    }

    // Encrypt using matrix multiplication
    vector<int> encryptedVec(SIZE, 0);
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            encryptedVec[i] += keyMatrix[i][j] * vec[j];
        }
        encryptedVec[i] %= 26;
        result += encryptedVec[i] + 'A';
    }

    return result;
}

int main() {
    // Key matrix (example)
    vector<vector<int>> keyMatrix = {
        {6, 24, 1},
        {13, 16, 10},
        {20, 17, 15}
    };

    string text = "ACT";  // Plaintext
    cout << "Ciphertext: " << hillCipher(text, keyMatrix);
    return 0;
}
```

Explanation:

1. **Matrix Inversion:** To decrypt, you need to find the modular inverse of the key matrix. This step is complex and often requires additional functions for determinant calculation and adjugate matrix. For simplicity, this code assumes you either precompute or use a library for matrix inversion.
2. **hillCipher:** This function converts the plaintext into numerical vectors, performs matrix multiplication with the key matrix, and then converts the result back into characters.

Note:

- This implementation is simplified and assumes the key matrix is 3×3. You need to handle matrix inversion in real scenarios or use a numerical library.

- For complete functionality, you would need to include the matrix inversion logic or use an existing library to compute it.

---

5. W.A.P. to Implement Rail Fence Technique

Concept:

The Rail Fence Cipher is a type of transposition cipher. It arranges the plaintext in a zigzag pattern across multiple "rails" (rows) and then reads the ciphertext row by row. This technique is simple but effective for basic encryption.

Algorithm:

1. **Choose the Number of Rails:** Decide how many rows (rails) will be used for the zigzag pattern.
2. **Write the Plaintext in Zigzag Pattern:**
    - Write the plaintext in a zigzag pattern down and up across the rails.
3. **Read Off Row by Row:**
    - Construct the ciphertext by concatenating the characters from each rail.

Example:

- **Plaintext:** "HELLO"
- **Number of Rails:** 3

  ```
  H . . . O
  . E . L .
  . . L . .
  ```

- **Ciphertext:** "HOEL LL"

C++ Code Implementation:

```
#include <iostream>
#include <vector>
using namespace std;
```

```
}
```

```cpp
// Function to encrypt plaintext using Rail Fence cipher
string railFenceEncrypt(const string& text, int rails) {
    if (rails <= 1) return text;  // No encryption if only one rail

    // Create a 2D array to store the rails
    vector<vector<char>> rail(rails, vector<char>(text.length(), '\\n'));

    int row = 0, dir = 1;  // Start from the first rail and move downwards
    for (int i = 0; i < text.length(); i++) {
        rail[row][i] = text[i];
        if (row == 0 || row == rails - 1) dir = -dir;  // Change direction at the ends
        row += dir;
    }

    // Read the ciphertext row by row
    string result = "";
    for (int i = 0; i < rails; i++) {
        for (int j = 0; j < text.length(); j++) {
            if (rail[i][j] != '\\n') result += rail[i][j];
        }
    }
    return result;
}

int main() {
    string text = "HELLO";  // Plaintext
    int rails = 3;  // Number of rails

    cout << "Ciphertext: " << railFenceEncrypt(text, rails);
    return 0;
}
```

Explanation:

1. **railFenceEncrypt:** This function arranges the plaintext in a zigzag pattern across the specified number of rails.
    - **row:** Tracks the current rail (row) position.
    - **dir:** Controls the direction of movement (down or up).
    - The zigzag pattern is constructed by placing characters at appropriate positions in the 2D rail array.
2. **Reading the Ciphertext:** The ciphertext is formed by concatenating characters from each rail row by row.

Note:

- This implementation assumes that the plaintext is short and fits into memory. For longer texts, you might need to manage memory more efficiently.
- The code handles basic cases. For complex cases, such as very long texts or non-alphabetic characters, additional handling might be required.

---

6. W.A.P. to Implement Simple Columnar Transposition Technique

Concept:

The Simple Columnar Transposition Cipher involves writing the plaintext in rows of a grid and then reading the columns in a specific order determined by a key. This is a form of transposition cipher where the order of characters is rearranged rather than altered.

Algorithm:

1. **Choose a Key:** The key determines the column order for reading the grid.
2. **Create the Grid:**
    - Write the plaintext into a grid with a fixed number of columns (determined by the key length).
3. **Read Columns in Key Order:**

- Read the columns based on the order of the key to form the ciphertext.

Example:

- **Plaintext:** "WEAREDISCOVERED"
- **Key:** "4312"
- **Grid:**

  ```
  W E A R
  E D I S
  C O V E
  R E D X
  ```

- **Column Order:** 4 3 1 2
- **Ciphertext:** "REOEDISCVXARE"

C++ Code Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to perform columnar transposition cipher
string columnarTranspositionEncrypt(const string& text, con
st string& key) {
    int numCols = key.length();
    int numRows = (text.length() + numCols - 1) / numCols;

    // Create a grid to store the text
    vector<vector<char>> grid(numRows, vector<char>(numCol
s, ' '));

    // Fill the grid with plaintext
    for (int i = 0; i < text.length(); i++) {
        int row = i / numCols;
        int col = i % numCols;
        grid[row][col] = text[i];
```

```
    }

    // Create a vector of column indices based on the key
    vector<int> colOrder(numCols);
    for (int i = 0; i < numCols; i++) {
        colOrder[i] = i;
    }
    sort(colOrder.begin(), colOrder.end(), [&key](int i, in
t j) {
        return key[i] < key[j];
    });

    // Read columns in the order defined by the key
    string result = "";
    for (int i : colOrder) {
        for (int row = 0; row < numRows; row++) {
            if (grid[row][i] != ' ') {
                result += grid[row][i];
            }
        }
    }
    return result;
}

int main() {
    string text = "WEAREDISCOVERED";  // Plaintext
    string key = "4312";  // Column order key

    cout << "Ciphertext: " << columnarTranspositionEncrypt
(text, key);
    return 0;
}
```

Explanation:

1. **Grid Creation:** The plaintext is written into a grid where the number of columns is determined by the length of the key.

2. **Column Ordering:** The columns are read in the order specified by the key, where each digit in the key represents a column index.
    - For the key "4312", columns are read in the order of 4th, 3rd, 1st, and 2nd columns.
3. **Reading Columns:** After sorting the column indices based on the key, characters are read from the grid in the specified column order to form the ciphertext.

Note:

- The implementation assumes that the plaintext length is a multiple of the key length or is padded to fit.
- This code handles basic cases; more complex handling might be required for very long texts or different key lengths.

---

7. W.A.P. to Implement Advanced Columnar Transposition Technique

Concept:

The Advanced Columnar Transposition Cipher extends the Simple Columnar Transposition by adding complexity. This can include multiple passes of transposition, padding, and more intricate column rearrangement. The basic idea is similar but with added complexity to increase security.

Algorithm:

1. **Choose a Key:** The key determines the column order for each pass.
2. **Create the Grid:**
    - Write the plaintext into a grid with a number of columns equal to the length of the key.
3. **Perform Multiple Passes:**
    - Rearrange columns based on the key, read off the columns, and repeat the process if required.
4. **Handle Padding:** Pad the plaintext if necessary to fit the grid size.

Example:

- **Plaintext:** "WEAREDISCOVERED"
- **Key 1:** "4312"
- **Key 2:** "2143" (for a second pass)

1. **First Pass:**
    - Grid (Key 1: "4312"):

      ```
      W E A R
      E D I S
      C O V E
      R E D X
      ```

    - Read columns in order: 4 3 1 2
    - Intermediate Ciphertext: "REOEDISCVXARE"
2. **Second Pass:**
    - Write Intermediate Ciphertext into a new grid using Key 2.
    - Rearrange and read columns again to get final ciphertext.

C++ Code Implementation:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to perform advanced columnar transposition ciph
er with multiple passes
string advancedColumnarTranspositionEncrypt(const string& t
ext, const string& key1, const string& key2) {
    int numCols = key1.length();
    int numRows = (text.length() + numCols - 1) / numCols;

    // Create the initial grid
    vector<vector<char>> grid(numRows, vector<char>(numCol
s, ' '));
```

```
    for (int i = 0; i < text.length(); i++) {
        int row = i / numCols;
        int col = i % numCols;
        grid[row][col] = text[i];
    }

    // Function to rearrange columns based on the key
    auto rearrangeColumns = [](vector<vector<char>>& grid,
const string& key) {
        vector<int> colOrder(grid[0].size());
        for (int i = 0; i < colOrder.size(); i++) {
            colOrder[i] = i;
        }
        sort(colOrder.begin(), colOrder.end(), [&key](int
i, int j) {
            return key[i] < key[j];
        });

        vector<vector<char>> newGrid(grid.size(), vector<ch
ar>(grid[0].size(), ' '));
        for (int i = 0; i < grid.size(); i++) {
            for (int j = 0; j < grid[0].size(); j++) {
                newGrid[i][j] = grid[i][colOrder[j]];
            }
        }
        grid = newGrid;
    };

    // First pass with key1
    rearrangeColumns(grid, key1);

    // Read columns and prepare intermediate ciphertext
    string intermediateCiphertext = "";
    for (int i = 0; i < numCols; i++) {
        for (int row = 0; row < numRows; row++) {
            if (grid[row][i] != ' ') {
                intermediateCiphertext += grid[row][i];
            }
        }
    }

    // Create a new grid for the second pass
    vector<vector<char>> newGrid(numRows, vector<char>(numC
ols, ' '));
    for (int i = 0; i < intermediateCiphertext.length(); i+
+) {
        int row = i / numCols;
        int col = i % numCols;
        newGrid[row][col] = intermediateCiphertext[i];
    }

    // Second pass with key2
    rearrangeColumns(newGrid, key2);

    // Read final ciphertext
    string finalCiphertext = "";
    for (int i = 0; i < numCols; i++) {
        for (int row = 0; row < numRows; row++) {
            if (newGrid[row][i] != ' ') {
                finalCiphertext += newGrid[row][i];
            }
        }
    }

    return finalCiphertext;
}

int main() {
    string text = "WEAREDISCOVERED";  // Plaintext
    string key1 = "4312";  // Key for the first pass
    string key2 = "2143";  // Key for the second pass

    cout << "Ciphertext: " << advancedColumnarTransposition
Encrypt(text, key1, key2);
    return 0;
}
```

Explanation:

1. **Grid Creation:** The plaintext is written into a grid based on the length of the first key.
2. **Rearrange Columns:** Columns are rearranged based on the key. This is done by sorting the column indices according to the key.
   - `rearrangeColumns`: This function rearranges the columns of the grid based on the provided key.
3. **Intermediate Ciphertext:** After the first pass, read the columns to get intermediate ciphertext.
4. **Second Pass:** Write the intermediate ciphertext into a new grid using the second key and rearrange columns again to get the final ciphertext.

Note:

- This implementation demonstrates a simple two-pass encryption. More passes or complex rearrangement can be added based on requirements.
- Ensure proper padding if the length of the text is not a perfect multiple of the key length.
