PASSWORD ENCODER AND DECODER

(Secure Password Management System using Huffman Encoding)

TEAM MEMBERS :

1. Gopala Krishnan D – 2022115079
2. Harish M - 2022115128
3. Sree Ram T R – 2022115108

PROBLEM STATEMENT:

A comprehensive Secure Password Management System that provides functionalities for encrypting and decrypting passwords using Huffman encoding, validating password strength, generating strong passwords, and determining the minimum edits required to convert a string into a palindrome.

SUMMARY :

This project implements a comprehensive menu-driven program focused on password encryption, validation, and various related functionalities. It leverages Huffman coding for efficient encryption and decryption, allowing users to securely encrypt their passwords. Additionally, the program provides features to check password validity and strength, generate strong passwords, and compute the minimum number of edits required to transform a string into a palindrome. The menu-driven interface ensures user-friendly interactions, enabling users to access and utilize these features conveniently.

DESIGN STRATEGY:

ALGORITHM USED :

- Huffman Coding Algorithm
- Dynamic Programming for Minimum Edits

TIME COMPLEXITY :

The project primarily employs two algorithms: Huffman Coding and Dynamic Programming for palindrome transformation.

HUFFMAN CODE ANALYSIS :

PURPOSE : Used to encrypt and decrypt passwords efficiently.

PROCESS :

1. **Build the Huffman Tree:**

    a. **Construct Frequency Map: Calculate the frequency of each character in the password. This takes (O(n)) time, where (n) is the length of the password.**

    b. **Build Priority Queue and Tree: Use a priority queue to build the Huffman Tree by repeatedly merging the two nodes with the smallest frequencies until only one node remains. This step involves inserting distinct nodes (distinct characters) into the priority queue, and each insertion operation takes (O(log d)) time, leading to an overall time complexity of (O(d log d)), where (d) is the number of distinct characters.**

2. Generate Huffman Codes: o Traverse the Huffman Tree to generate binary codes for each character. This takes O(d)O(d)O(d) me, where d is the number of dis nct characters.

DYNAMIC PROGRAMMING FOR MINIMUM EDITS TO PAINDROME :

PURPOSE : Determine the minimum number of edits needed to convert a string into a palindrome.

PROCESS :

- Use a dynamic programming table to store the lengths of the longest palindromic subsequences for various substrings of the input string.
- Fill the table iteratively, where each cell dp[i][j] represents the length of the longest palindromic subsequence within the substring s[i..j]

TIME COMPLEXITY ANALYSIS :

- Filling the dynamic programming table involves iterating over all substrings of the string, leading to a time complexity of O(n^2), where n is the length of the string.

EXAMPLE CALCULATION :

- Suppose we have a string of length n=10.
    - Filling the dynamic programming table: O(10^2) = O(100)

ALGORITHM :

HUFFMAN CODE :

```
Function buildHuffmanTree(freq_map):
    Create a min-heap (priority queue) PQ
    for each character c in freq_map:
        Create a node with character c and frequency freq_map[c]
        Insert node into PQ

    while PQ.size() > 1:
        left = PQ.extractMin()
        right = PQ.extractMin()
        newNode = Node(null, left.freq + right.freq)
        newNode.left = left
        newNode.right = right
        PQ.insert(newNode)

    return PQ.extractMin() # Root of Huffman Tree

Function generateHuffmanCodes(root, code, huffmanCode):
    if root is None:
        return
    if root.char is not null:
        huffmanCode[root.char] = code
    generateHuffmanCodes(root.left, code + "0", huffmanCode)
    generateHuffmanCodes(root.right, code + "1", huffmanCode)

Function encryptPassword(password, huffmanCode):
    encryptedPassword = ""
    for each character c in password:
        encryptedPassword += huffmanCode[c]
    return encryptedPassword

Function decryptPassword(encryptedPassword, root):
    decryptedPassword = ""
    currentNode = root
    for each bit in encryptedPassword:
        if bit == '0':
            currentNode = currentNode.left
        else:
            currentNode = currentNode.right
        if currentNode.char is not null:
            decryptedPassword += currentNode.char
            currentNode = root
    return decryptedPassword
```

DYNAMIC PROGRAMMING :

```
Function minEditsToPalindrome(s):
    n = length of s
    DP = 2D array of size n x n initialized to 0

    for i from 0 to n-1:
        DP[i][i] = 1

    for length from 2 to n:
        for i from 0 to n-length:
            j = i + length - 1
            if s[i] == s[j]:
                DP[i][j] = DP[i+1][j-1] + 2
            else:
                DP[i][j] = max(DP[i+1][j], DP[i][j-1])

    lpsLength = DP[0][n-1]
    minEdits = n - lpsLength
    return minEdits
```

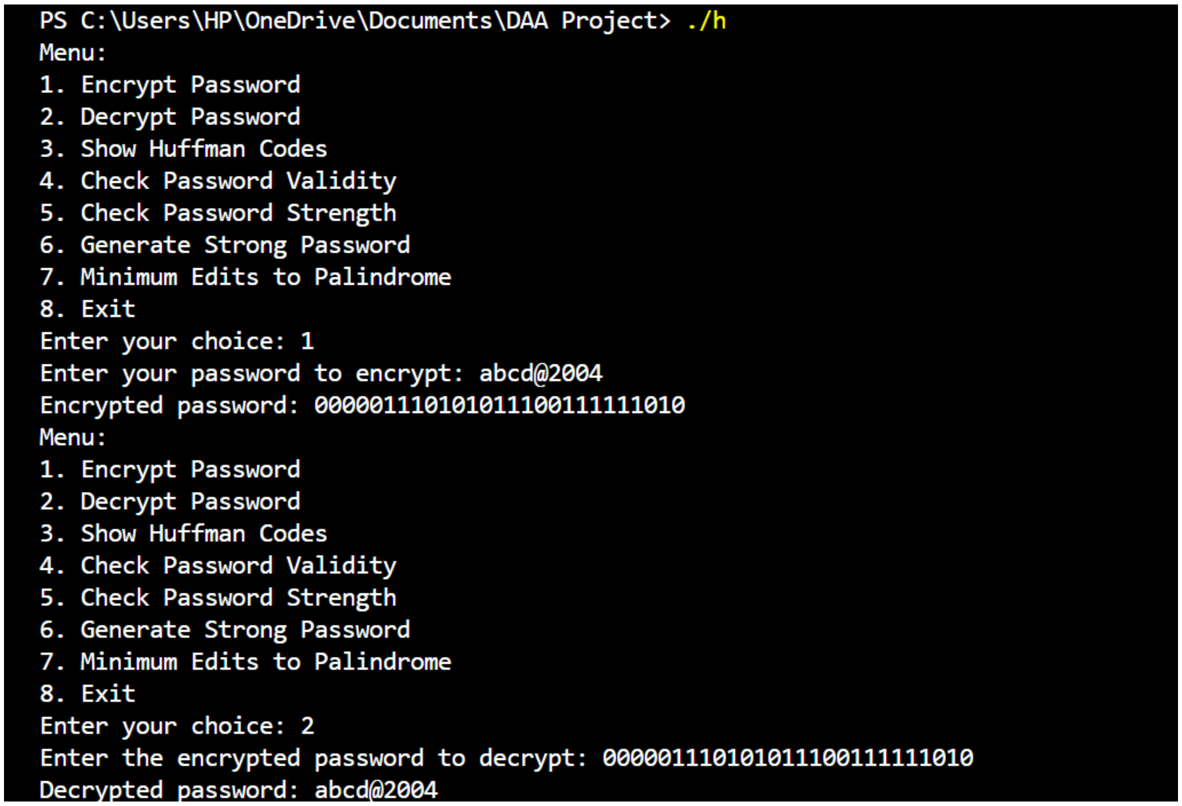
CODE :

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <queue>
#include <random>
#include <regex>

using namespace std;

// Huffman Tree Node
struct Node
{
    char ch;
    int freq;
    Node *left, *right;

    Node(char ch, int freq) : ch(ch), freq(freq), left(nullptr),
right(nullptr) {}
};

struct Compare
{
    bool operator()(Node *left, Node *right)
    {
        return left->freq > right->freq;
    }
};

// Function to build the Huffman Tree
Node *buildHuffmanTree(const unordered_map<char, int> &freq)
{
    priority_queue<Node *, vector<Node *>, Compare> minHeap;

    for (const auto &pair : freq)
    {
        minHeap.push(new Node(pair.first, pair.second));
    }

    while (minHeap.size() > 1)
    {
        Node *left = minHeap.top();
        minHeap.pop();
        Node *right = minHeap.top();
        minHeap.pop();

        int sum = left->freq + right->freq;
        Node *newNode = new Node('\0', sum);
        newNode->left = left;
        newNode->right = right;
        minHeap.push(newNode);
    }

    return minHeap.top();
}

// Function to generate Huffman Codes
void generateHuffmanCodes(Node *root, const string &str, unordered_map<char,
string> &huffmanCode)
{
    if (!root)
        return;
    if (root->ch != '\0')
    {
        huffmanCode[root->ch] = str;
    }

    generateHuffmanCodes(root->left, str + "0", huffmanCode);
    generateHuffmanCodes(root->right, str + "1", huffmanCode);
}

// Function to encrypt the password using Huffman Codes
string encryptPassword(const string &password, const unordered_map<char,
string> &huffmanCode)
{
    string encryptedPassword;
    for (char ch : password)
    {
        encryptedPassword += huffmanCode.at(ch);
    }
    return encryptedPassword;
}

// Function to decrypt the encrypted password using Huffman Tree
string decryptPassword(const string &encryptedPassword, Node *root)
{
    string decryptedPassword;
    Node *currNode = root;
    for (char bit : encryptedPassword)
    {
        if (bit == '0')
        {
            currNode = currNode->left;
        }
        else
        {
            currNode = currNode->right;
        }
        if (currNode->ch != '\0')
        {
            decryptedPassword += currNode->ch;
            currNode = root;
        }
    }
    return decryptedPassword;
}

// Function to delete the Huffman Tree nodes
void deleteHuffmanTree(Node *root)
{
    if (!root)
        return;
    deleteHuffmanTree(root->left);
    deleteHuffmanTree(root->right);
    delete root;
}

// Function to check if the password meets length criteria
bool isPasswordValid(const string &password, int minLength, int maxLength)
{
    int length = password.length();
    return (length >= minLength && length <= maxLength);
}

// Function to check password strength
bool isStrongPassword(const string &password)
{
    if (password.length() < 8)
    {
        return false;
    }
    bool hasUpper = false, hasLower = false, hasDigit = false, hasSpecial =
false;
    for (char ch : password)
    {
        if (isupper(ch))
            hasUpper = true;
        else if (islower(ch))
            hasLower = true;
        else if (isdigit(ch))
            hasDigit = true;
        else
            hasSpecial = true;
    }
    return hasUpper && hasLower && hasDigit && hasSpecial;
}

// Function to generate a random strong password
string generatePassword(int length)
{
    const string characters =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()";
    random_device rd;
    mt19937 generator(rd());
    uniform_int_distribution<> distribution(0, characters.size() - 1);

    string password;
    for (int i = 0; i < length; ++i)
```

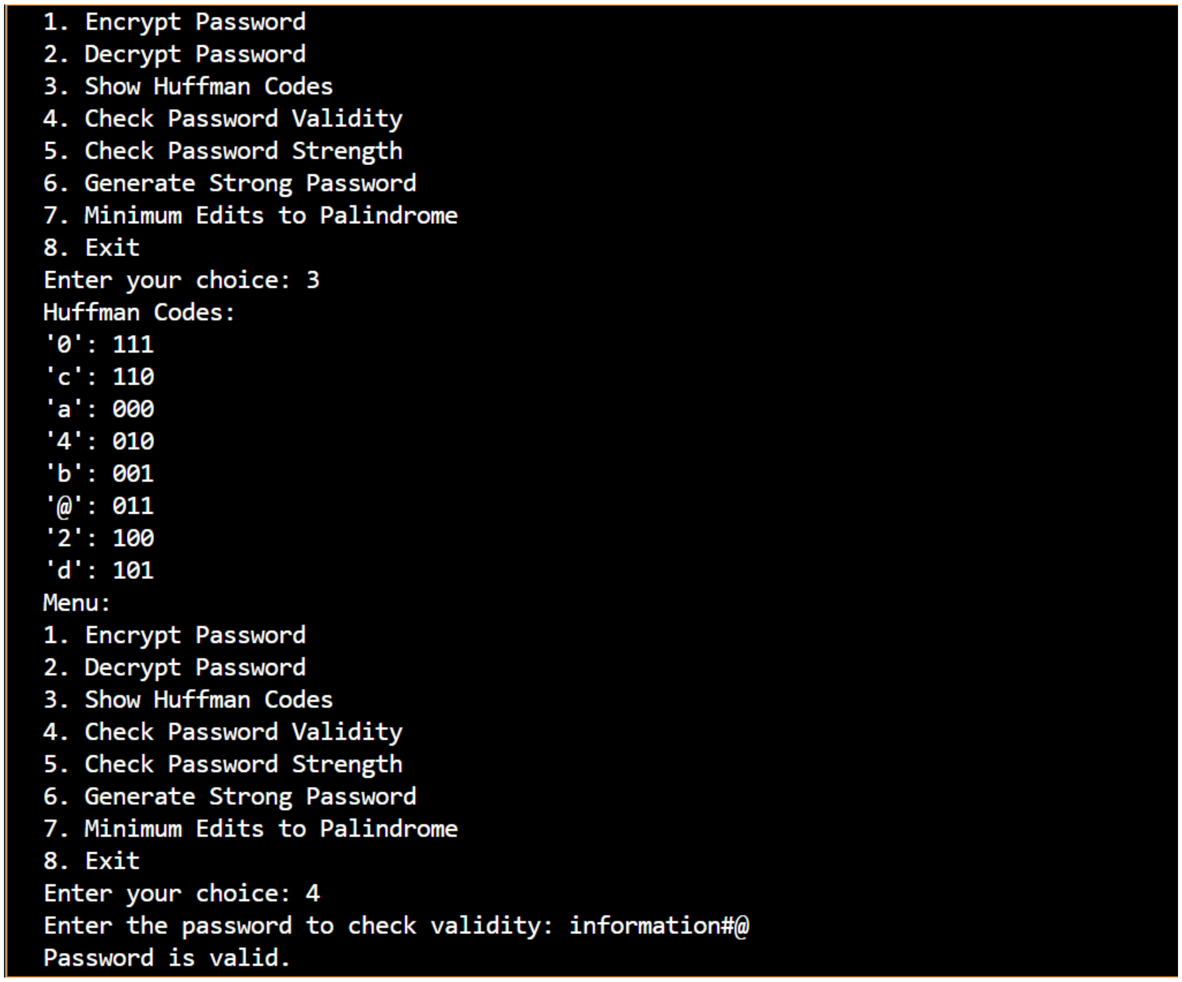
```cpp
    {
        password += characters[distribution(generator)];
    }

    return password;
}

// Function to find minimum edits to make a string palindrome
int minEditsToPalindrome(const string &s)
{
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++)
    {
        dp[i][i] = 1;
    }
    for (int length = 2; length <= n; length++)
    {
        for (int i = 0; i <= n - length; i++)
        {
            int j = i + length - 1;
            if (s[i] == s[j])
            {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            }
            else
            {
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }

    int lpsLength = dp[0][n - 1];
    int minEdits = n - lpsLength;

    return minEdits;
}

int main()
{
    char choice;
    string password;
    Node *huffmanRoot = nullptr; // Pointer to the Huffman tree root

    // Menu-driven interface
    while (true)
    {
        cout << "Menu:" << endl;
        cout << "1. Encrypt Password" << endl;
        cout << "2. Decrypt Password" << endl;
        cout << "3. Show Huffman Codes" << endl;
        cout << "4. Check Password Validity" << endl;
        cout << "5. Check Password Strength" << endl;
        cout << "6. Generate Strong Password" << endl;
        cout << "7. Minimum Edits to Palindrome" << endl;
        cout << "8. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice)
        {
        case '1':
        {
            if (huffmanRoot)
            {
                deleteHuffmanTree(huffmanRoot);
            }
            cout << "Enter your password to encrypt: ";
            cin >> password;

            if (!isPasswordValid(password, 6, 20))
            {
                cout << "Password length should be between 6 and 20
characters." << endl;
                break;
            }

            unordered_map<char, int> freq;
            for (char ch : password)
            {
                freq[ch]++;
            }

            huffmanRoot = buildHuffmanTree(freq);

            unordered_map<char, string> huffmanCode;
            generateHuffmanCodes(huffmanRoot, "", huffmanCode);

            string encryptedPassword = encryptPassword(password, huffmanCode);
            cout << "Encrypted password: " << encryptedPassword << endl;

            break;
        }
        case '2':
        {
            if (!huffmanRoot)
            {
                cout << "No password has been encrypted yet." << endl;
                break;
            }
            cout << "Enter the encrypted password to decrypt: ";
            cin >> password;

            string decryptedPassword = decryptPassword(password, huffmanRoot);
            cout << "Decrypted password: " << decryptedPassword << endl;

            break;
        }
        case '3':
        {
            if (!huffmanRoot)
            {
                cout << "No Huffman tree available." << endl;
                break;
            }
            cout << "Huffman Codes:" << endl;
            unordered_map<char, string> huffmanCode;
            generateHuffmanCodes(huffmanRoot, "", huffmanCode);
            for (const auto &pair : huffmanCode)
            {
                cout << "'" << pair.first << "': " << pair.second << endl;
            }
            break;
        }
        case '4':
        {
            cout << "Enter the password to check validity: ";
            cin >> password;
            if (isPasswordValid(password, 6, 20))
            {
                cout << "Password is valid." << endl;
            }
            else
            {
                cout << "Password is invalid. Password length should be
between 6 and 20 characters." << endl;
            }
            break;
        }
        case '5':
        {
            cout << "Enter a password to check its strength: ";
            cin >> password;
```

```cpp
            if (isStrongPassword(password))
            {
                cout << "Password is strong." << endl;
            }
            else
            {
                cout << "Password is weak. It should be at least 8 characters
long, and include uppercase letters, lowercase letters, digits, and special
characters." << endl;
            }
            break;
        }
        case '6':
        {
            int length;
            cout << "Enter the desired length for the password: ";
            cin >> length;

            string password = generatePassword(length);
            cout << "Generated password: " << password << endl;

            break;
        }
        case '7':
        {
            cout << "Enter the string to check minimum edits to palindrome: ";
            cin >> password;
            int edits = minEditsToPalindrome(password);
            cout << "Minimum edits needed to make the string a palindrome: "
<< edits << endl;
            break;
        }
        case '8':
        {
            if (huffmanRoot)
            {
                deleteHuffmanTree(huffmanRoot);
            }
            cout << "Exiting the program." << endl;
            return 0;
        }
        default:
        {
            cout << "Invalid choice. Please enter a valid option." << endl;
        }
        }
    }
    return 0;
}
```

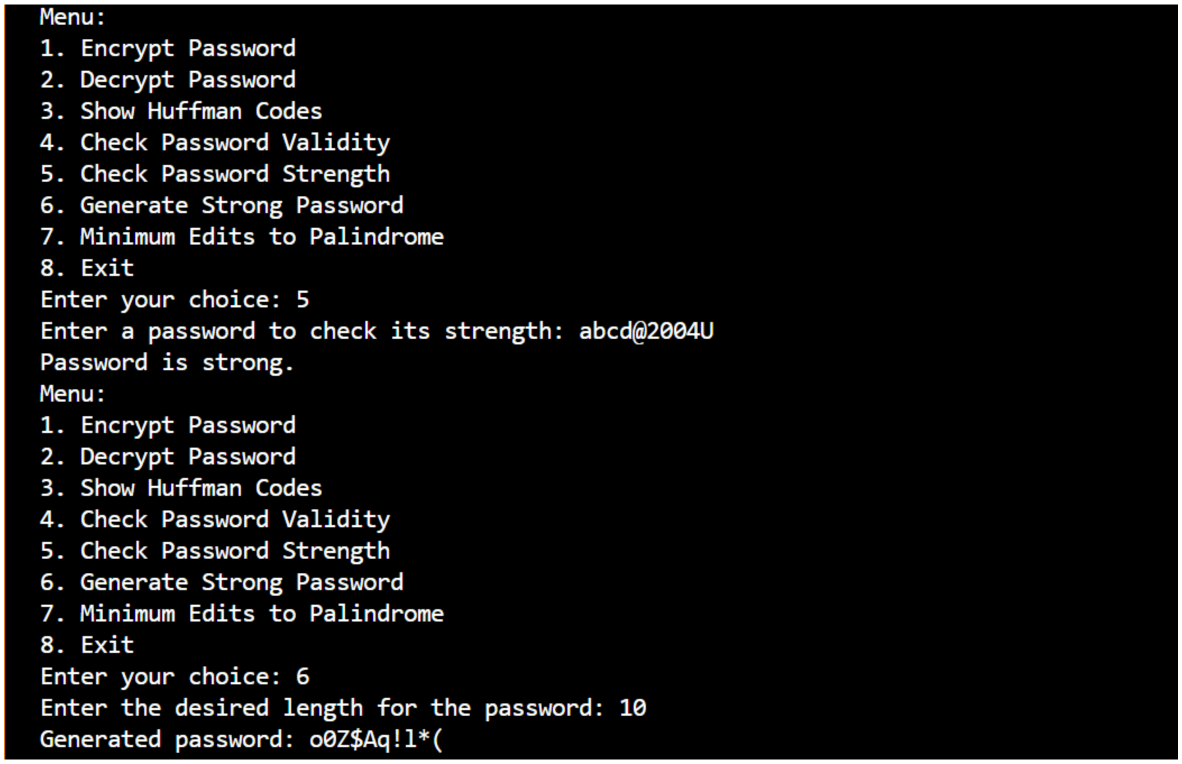
OUTPUT:



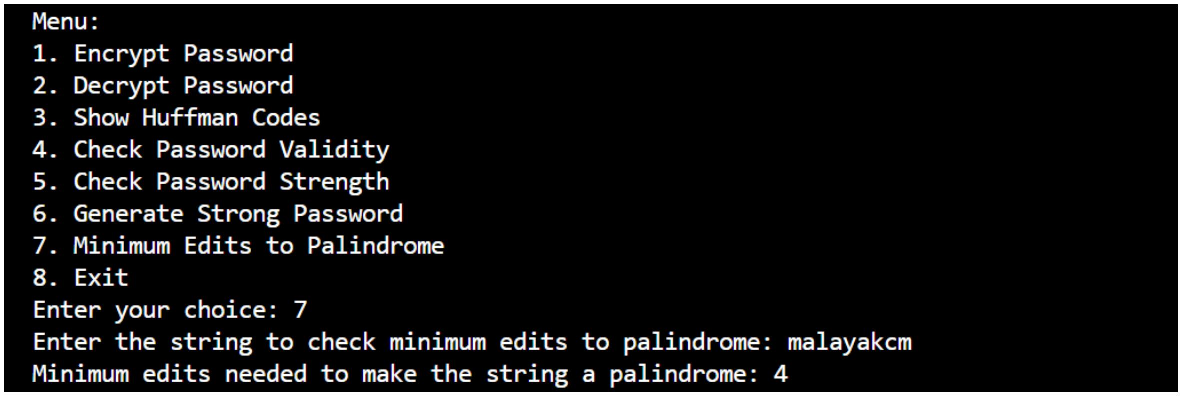
```
PS C:\Users\HP\OneDrive\Documents\DAA Project> ./h
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 1
Enter your password to encrypt: abcd@2004
Encrypted password: 000001110101011100111111010
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 2
Enter the encrypted password to decrypt: 000001110101011100111111010
Decrypted password: abcd@2004
```



```
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 3
Huffman Codes:
'0': 111
'c': 110
'a': 000
'4': 010
'b': 001
'@': 011
'2': 100
'd': 101
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 4
Enter the password to check validity: information#@
Password is valid.
```



```
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 5
Enter a password to check its strength: abcd@2004U
Password is strong.
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 6
Enter the desired length for the password: 10
Generated password: o0Z$Aq!l*(
```



```
Menu:
1. Encrypt Password
2. Decrypt Password
3. Show Huffman Codes
4. Check Password Validity
5. Check Password Strength
6. Generate Strong Password
7. Minimum Edits to Palindrome
8. Exit
Enter your choice: 7
Enter the string to check minimum edits to palindrome: malayakcm
Minimum edits needed to make the string a palindrome: 4
```

Conclusion:

The Secure Password Management System offers robust password protection through Huffman encoding for encryption and decryption. It ensures password validity and strength, generates strong passwords, and computes minimum edits for palindrome conversion. With a user-friendly interface and focus on performance and security, it's a comprehensive solution for secure password management, promoting data integrity and user security.