# Adaptive Huffman Algorithm

Presented By:
Harshul Jain (231IT025)
Nikhil Agarwal(231IT044)

# Abstract


- This project presents an enhanced data compression technique using the Adaptive Huffman Algorithm, combined with text clustering and multiple character modification, to improve compression efficiency for textual data
- By grouping similar sequences and reducing redundancy, the modified algorithm achieves superior compression ratios compared to traditional methods

# Introduction

- Data compression techniques aim to reduce the storage size of data without compromising data integrity. Adaptive Huffman coding, a popular lossless compression algorithm, dynamically adjusts based on the input data stream, allowing high compression ratios. However, it encounters limitations with repetitive or similar patterns in textual data.
- This study introduces text clustering and multiple character modification to the Adaptive Huffman Algorithm to address these limitations. By identifying and clustering repetitive patterns, the algorithm compresses data more effectively, achieving a better compression ratio than conventional techniques.

# Methodology

This study's methodology involves modifying the Adaptive Huffman algorithm by integrating:

1. **Text Clustering**: Identification of frequent clusters within the encoded data using pattern recognition techniques.
2. **Character Modification**: Replacing identified clusters with new characters, adjusting the Huffman tree accordingly.

```cpp
#include <bits/stdc++.h>
using namespace std;
// TreeNode structure for the Huffman Tree
struct TreeNode {
    char character;
    int frequency;
    TreeNode *left, *right;

    TreeNode(char character, int frequency) {
        left = right = nullptr;
        this->character = character;
        this->frequency = frequency;
    }
};

// Comparator to sort TreeNodes by frequency in the priority queue
struct cmp {
    bool operator()(TreeNode* left, TreeNode* right) {
       return left->frequency > right->frequency;
    }
};

// Traverse the Huffman tree and store the Huffman codes in a map
void encode(TreeNode* root, string str, unordered_map<char, string> &huffmanCode) {
    if (!root) return;

    if (!root->left && !root->right) {
        huffmanCode[root->character] = str;
    }

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

```cpp
// Build the Huffman Tree and return the root
TreeNode* buildHuffmanTree(const unordered_map<char, int>& freq) {
    priority_queue<TreeNode*, vector<TreeNode*>, cmp> pq;

    // Create a leaf TreeNode for each character and add it to the priority queue
    for (auto pair : freq) {
        pq.push(new TreeNode(pair.first, pair.second));
    }

    // Iterate until there is only one TreeNode in the priority queue
    while (pq.size() != 1) {
        // Remove two TreeNodes of the highest priority (lowest frequency)
        TreeNode *left = pq.top(); pq.pop();
        TreeNode *right = pq.top(); pq.pop();

        // Create a new internal TreeNode with a frequency equal to the sum of the two TreeNodes
        TreeNode* sum = new TreeNode('\0', left->frequency + right->frequency);
        sum->left = left;
        sum->right = right;

        pq.push(sum);
    }

    // The remaining TreeNode is the root of the Huffman Tree
    return pq.top();
}

// Function to calculate frequencies of characters in the input text
unordered_map<char, int> calculateFrequency(const string& text) {
    unordered_map<char, int> freq;
    for (char ch : text) {
        freq[ch]++;
    }
```

```cpp
// Function to encode input text using the generated Huffman codes
string encodeText(const string& text, const unordered_map<char, string>& huffmanCode) {
    string encodedString = "";
    for (char ch : text) {
        encodedString += huffmanCode.at(ch);
    }
    return encodedString;
}

// Improved Text Clustering based on frequency analysis and longer patterns
void textClustering(string& encodedText, unordered_map<string, char>& patternMap, char& nextReplacement) {
    unordered_map<string, int> patternFrequency;

    // Find longer patterns (e.g., 4 or 5 characters) and count their occurrences
    for (size_t length = 4; length <= 5; ++length) {  // Try 4 or 5 character patterns
        for (size_t i = 0; i < encodedText.length() - length + 1; ++i) {
            string pattern = encodedText.substr(i, length);
            patternFrequency[pattern]++;
        }
    }

    // Replace frequent patterns with new characters (apply a higher threshold)
    for (const auto& entry : patternFrequency) {
        if (entry.second > 3) {  // Adjust threshold as needed (e.g., > 3)
            string pattern = entry.first;
            char replacement = nextReplacement++;
            patternMap[pattern] = replacement;

            size_t pos = encodedText.find(pattern);
            while (pos != string::npos) {
                encodedText.replace(pos, pattern.length(), string(1, replacement));
                pos = encodedText.find(pattern, pos + 1);
            }
```

```cpp
// Modify the frequency table to include new replacement characters
unordered_map<char, int> modifyFrequenciesWithPattern(const string& text, const unordered_map<string, char>& patternMap) {
    unordered_map<char, int> freq;
    for (char ch : text) {
        freq[ch]++;
    }

    // Add frequencies of newly introduced replacement characters
    for (const auto& entry : patternMap) {
        freq[entry.second]++;
    }

    return freq;
}

// Huffman Compression Function
void huffmanCompression(const string& text) {
    // Step 1: Calculate frequencies of characters
    unordered_map<char, int> freq = calculateFrequency(text);

    // Step 2: Build the Huffman Tree
    TreeNode* root = buildHuffmanTree(freq);

    // Step 3: Generate Huffman Codes
    unordered_map<char, string> huffmanCode;
    encode(root, "", huffmanCode);

    // Step 4: Encode the input text using Huffman Codes
    string encodedText = encodeText(text, huffmanCode);
    cout << "Encoded Text: " << encodedText << endl;
    cout << "Encoded Text length: " << encodedText.length() << endl;
```

```
    // Step 5: Apply improved text clustering
    unordered_map<string, char> patternMap;
    char nextReplacement = 'A'; // Start replacing patterns with characters like 'A', 'B', etc.
    textClustering(encodedText, patternMap, nextReplacement);

    // Step 6: Modify frequency table with new patterns and rebuild Huffman Tree
    freq = modifyFrequenciesWithPattern(encodedText, patternMap);
    root = buildHuffmanTree(freq);

    // Step 7: Re-encode the modified text
    huffmanCode.clear();
    encode(root, "", huffmanCode);
    encodedText = encodeText(encodedText, huffmanCode);
    cout << "Re-encoded Text after Clustering: " << encodedText << endl;
    cout << "Re-encoded Text length after clustering: " << encodedText.length() << endl;
}

int main() {
    string text = "aaaaabbbbbccccaaaaabbbbcccddddddaaaabbbbccccc";

    // Perform Huffman Compression
    huffmanCompression(text);

    return 0;
}
```

# Output

```
Encoded Text: 1111111111101010101001010101111111111111010101001010100000000011111111101010100101010101
Encoded Text length: 88
Re-encoded Text after Clustering: 1101100001110101000011110110101000100011101110011011110101101101000100011111
Re-encoded Text length after clustering: 76
```

# Setup

- Environment: The algorithm was developed and tested using C++ on a standard computing setup having gcc compiler
- Tools and Structure: Several data structures and algorithms were used in writing the code for the project like

  Binary Tree
  Priority Queue
  Vector

# Performance Metrics

Compression ratio

execution time

**Output:**

**Original Huffman Algorithm Output**

**Original Encoded Text length: 72**

**Original Huffman Compression Time: 0.0021 seconds**

**Modified Huffman Algorithm with Clustering Output**

**Modified Encoded Text length after clustering: 64**

**Modified Huffman Compression Time: 0.0045 seconds**

# Performance Metrics (Contd.)

- Compression Ratio : Measured by taking ratio of Encoded text length with Re-encoded text length using modified Huffman Text compression using clustering techniques
- Execution Time: The execution time for the adaptive Huffman technique is slightly more than the conventional Huffman algorithm due to extra clustering operations.
- The execution time was measured using the std::<chrono> library of C++ which gives a precise measure of time taken by the code operations.

# Conclusion and Future Work

- The enhanced Adaptive Huffman Algorithm with text clustering and multiple character modification effectively compresses textual data, especially those with repetitive patterns. Future research could explore dynamic clustering thresholds and real-time optimization techniques for further efficiency.
- Additionally, the algorithm could be extended to non-textual data, adapting clustering and modification for various data types.