VIET NAM NATIONAL UNIVERSITY - HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

REPORT
STATIC HUFFMAN COMPRESSION

GROUP 4

Theory Lecturer	Dr.	Nguyen Thanh Phuong
Lab Lecturer	Dr.	Nguyen Ngoc Thao
	B.S.	Nguyen Thanh Tinh
Students	24127104	Du Hoai Phuc - Leader
	24127008	Truong Nhat Phat
	24127173	Nguyen Tuan Hung

Contents

General information

Member's information:

ID	Full name	Email	Roles
24127104	Du Hoai Phuc	dhphuc2424@clc.fitus.edu.vn	Leader, Report Editor
24127173	Nguyen Tuan Hung	nthung2432@clc.fitus.edu.vn	Data Analysis, Code Optimizer
24127008	Truong Nhat Phat	tnphat2418@clc.fitus.edu.vn	Algorithm Analysis, Editor

Table 1: Group's information

Task assignment

No.	Task	Assignee	Progress
1	Coding	Nguyen Tuan Hung	100%
2	Introduction to Compression	Nguyen Tuan Hung	100%
3	Introduction to Huffman Coding	Truong Nhat Phat	100%
4	Algorithm Theory	Truong Nhat Phat	100%
5	Algorithm Visualization	Du Hoai Phuc	100%
6	Algorithm variances	Truong Nhat Phat	100%
7	Performance Experiment	Nguyen Tuan Hung	100%
8	Pros and Cons	Nguyen Tuan Hung	100%
9	Discover the Application of Algorithm	Truong Nhat Phat	100%
10	Compare with other compression Algorithms	Nguyen Tuan Hung	100%
11	Check progress	Du Hoai Phuc	100%
12	Report combination	Du Hoai Phuc	100%
13	Record description	Du Hoai Phuc	100%
14	Report editing	Du Hoai Phuc	100%
15	Video editing	Truong Nhat Phat	100%
16	Slide	Du Hoai Phuc	100%

Table 2: Task assignment

1 Introduction to Compression

1.1 Overview of Compression

Data is a crucial part of the digital world, representing information in various forms such as text, images, audio, and video. It serves as the backbone of modern computing, communication, analysis, and decision-making across several industries. The rapid growth of data in recent years, has led to the development of advanced storage, processing, and compression techniques to optimize efficiency and accessibility. Proper data management is essential for ensuring security, accuracy, and usability in applications ranging from AI to cloud computing.
Data compression is a type of data management that helps reduce the size of data without (or with minimal) loss of information. In the era of G.AI where a large amount of data can be generated in a blink of an eye, data compression is not just desirable but an absolute necessity.

1.2 History of Compression

The history of data compression dates back to the early days of computing, driven by the need to store and transmit data efficiently since the storage device and bandwidth are limited resources. One of the first cases of data compression is Morse code which assigns shorter code to frequent use letters. In 1948, Claude Shannon laid the foundation for data compression with their groundbreaking work on information theory, introducing the concept of entropy and establishing the theoretical limits on lossless compression. In 1952, Huffman Coding was introduced by David A. Huffman, an optimal algorithm for generating prefix-free, variable-length codes. The 1970s and 80s brought the Lempel-Ziv family of algorithms (LZ77, LZ78, and LZW), which introduced dictionary-based methods and became widely adopted in various applications. The 1990s saw the rise of JPEG and MP3, which introduced lossy compression techniques for images and audio, balancing quality and file size. Today, data compression research continues to advance, exploring areas like deep learning-based compression and specialized algorithms for emerging data types, reflecting the ongoing quest to manage the ever-expanding volume of digital information.

1.3 Type of Compression

Data compression techniques can be broadly classified into two fundamental categories: lossless and lossy compression.

- **Lossless Compression**: These techniques guarantee perfect reconstruction of the original data after decompression. No information is lost in the process. Lossless compression is essential for applications where data integrity is paramount, such as text documents, executable files, and medical images. Examples include Run-Length Encoding (RLE), Huffman Coding, and the Lempel-Ziv family of algorithms.

- **Lossy Compression**: These techniques achieve higher compression ratios by discarding some information that is deemed less important or imperceptible. The decompressed data is an approximation of the original, but the loss is often acceptable in applications like image, audio, and video streaming, where minor imperfections are tolerable. Examples include JPEG for images, MP3 for audio, and MPEG for video.

Another distinction is between symmetric and asymmetric compression. Symmetric compression algorithms have similar computational complexity for both encoding and decoding, while asymmetric algorithms have significantly different complexities (typically, encoding is much more complex than decoding).

1.4 Lossless Compression

A. Run-Length Encoding (RLE)

- Basic principles: replacing sequences of identical data with a count and the repeated value.
- Example: "AAAAABBBCC" -> "5A3B2C"
- Application: Simple images, fax, text with repetitive patterns.
- Advantages: Simple, fast.
- Disadvantages: Not effective for data with little repetition.

B. Huffman Coding

- Basic principle: variable-length codes based on symbol frequencies (shorter codes for more frequent symbols).
- Example: "BEEBE BEE" -> 0111011000111 (in bit representation)
- Application: JPEG, MP3, GZIP, general-purpose compression.
- Advantages: Optimal for symbol-by-symbol encoding, widely used.
- Disadvantages: Requires frequency information (static) or adaptation (adaptive).

C. Arithmetic Coding

- Principle: Encodes the entire message as a single fraction within the range [0, 1). Subdivides the range based on symbol probabilities.
- Example: “ABBC” -> 0.140625
- Applications: JPEG 2000, JBIG2, high-performance compression.
- Advantages: Often achieves better compression than Huffman coding, especially for small alphabets.
- Disadvantages: More complex to implement, computationally more expensive.

D. Dictionary-Based Methods (Lempel-Ziv Family)

- LZ77:
  - Principle: Replaces repeated substrings with references to earlier occurrences within a sliding window.
  - Applications: GZIP, DEFLATE (used in ZIP, PNG).
- LZ78:
  - Principle: Builds a dictionary of previously seen substrings dynamically.
- LZW (Lempel-Ziv-Welch):
  - Principle: A variation of LZ78 that is widely used.
  - Applications: GIF, TIFF, early Unix "compress" utility.
- Advantages: Adaptive, good for text and data with repeating patterns.
- Disadvantages: Can be slower than simpler methods.

E. Burrows-Wheeler Transform (BWT)

- Principle: Reorders the input data to group similar characters together, making it more compressible by other methods (e.g., RLE, Move-to-Front, Huffman). It's a reversible transformation.
- Applications: bzip2 (high-performance compression).
- Advantages: Can achieve very high compression ratios, especially for text.
- Disadvantages: Relatively slow.

F. Others

- Prediction by Partial Matching (PPM)
- Context Tree Weighting (CTW)

1.5 Lossy Compression

A. Quantization

- Principle: Reducing the precision of data by mapping a range of values to a single representative value.
- Applications: Fundamental to most lossy compression methods.

B. Transform Coding

- Principle: Transforming the data into a different domain (e.g., frequency domain) where it can be more efficiently represented and quantized.
- Discrete Cosine Transform (DCT):
  - Principle: Widely used in image and video compression. Decomposes the data into different frequency components.
  - Applications: JPEG, MPEG, H.264.
- Discrete Wavelet Transform (DWT):
  - Principle: Provides a multi-resolution representation of the data.
  - Applications: JPEG 2000, image and video compression.
- Advantages: Allows for significant compression by discarding less important frequency components.
- Disadvantages: Introduces irreversible loss of information.

C. Vector Quantization

- Principle: Groups data into vectors and represents each vector with a codeword from a codebook.
- Applications: Image and speech compression.

D. Fractal Compression

- Principle: Exploits self-similarity in images to represent parts of the image with transformations of other parts.
- Applications: Image compression (historically more popular).

1.6 Application

A. Multimedia compression

- Image compression: JPEG, JPEG 2000, GIF, PNG, WebP.
- Audio compression: MP3, AAC, FLAC, Opus, Vorbis.
- Video compression: MPEG-1, MPEG-2, MPEG-4, H.264 (AVC), H.265 (HEVC), AV1, VP9.

B. Text compression

- GZIP, bzip2, general-purpose archiving tools

C. Others

- File system compression
- Backup and Archiving.
- Database Compression
- Network Compression

1.7 Conclusion

The future of data compression is promised to be driven by artificial intelligence, quantum computing. Machine learning based algorithms are already showing promise, using neural networks to understand the patterns in data to achieve a better compression rate than traditional methods. Autoencoders and generative models are being explored to learn more efficient data representations, adapting to the specific characteristics of different data types. New compression techniques now include encryption, making data both smaller and more secure, which is vital given rising cybersecurity concerns. As data continues to expand, the field of data compression will remain vital, constantly evolving to find new and more effective ways to store and transmit information in an increasingly data-driven world.

2 Introduction to Huffman coding

2.1 David Albert Huffman (1925–1999)

- David Albert Huffman came from Ohio, so he attended Ohio State University for his Bachelor's in electrical engineering. He earned it in 1944, when he was 18. After serving in the United States Navy, he returned to Ohio and studied for a Doctor of Philosophy in electrical engineering at MIT (1953).
- He made his only career move when he went to the University of California in Santa Cruz as the founding faculty member of the Computer Science Department in 1967. Huffman played a major role in the development of the department, he served as chair from 1970 to 1973. He is known for his motto "My products are my students.". In 1994, He retired and remained active in the department by teaching information theory and signal analysis courses.[5]
- Huffman has made significant contributions in several areas, primarily information theory and coding, signal design for radar and communications, and design procedures for asynchronous logic circuits. The Huffman coding algorithm is widely used to compress data, especially in the coding field.

2.2 The birth of Huffman coding

- When David Huffman studied for a Doctor of Philosophy at MIT and took a course on Information Theory taught by Robert M. Fano, Huffman had to do the final assignment for the course to find the most efficient way to encode a set of symbols based on their frequencies.
- At that time, Shannon-Fano coding was a popular algorithm, but it did not always bring the most efficient encoding. All the students in that class had to find an optimal encoding method or prove that no better method existed.
- Huffman struggled with complex mathematical approaches, attempting to optimize coding schemes using different strategies for weeks. He was about to give up but suddenly realized that instead of assigning arbitrary binary codes, he could build a binary tree from the bottom by repeatedly merging the two least frequent symbols. This technique can ensure that the most frequent symbols will be encoded with shorter code and minimize the length of bits needed to store data.
- Huffman coding was published in the paper "A Method for the Construction of Minimum-Redundancy Codes" in 1952.

2.3 Brief description of the algorithm

- Huffman coding is a popular algorithm in lossless data compression. This means it can compress data to reduce file sizes without sacrificing any significant information and the original data can be reconstructed perfectly in decoding. The algorithm assigns each input symbol using variable-length (binary) codes. These codes are based on the frequencies of corresponding symbols and are Prefix codes.
- Prefix codes are a type of code that a symbol when assigned by this code is not a prefix of code assigned to another symbol. For example, [00,11,10,010] is an array of prefix codes but [00,001,10,010] is not because 00 is a prefix of 001.

- Based on the description of the prefix code, Huffman coding ensures that each symbol will be kept intact in encoding and restores the original symbol in decoding. Let's take an example when the code of each symbol is not a prefix code:
- Let there be three characters x, y, z and their code be 0, 01, 00. The code assigned to x is the prefix of codes assigned to y and z. If the compressed bit is 00001, the de-compressed output may be "zxy" or "xzy".
- There are two main parts of Huffman coding:
  - Build a Huffman Tree from input
  - Traverse the Huffman Tree and assign codes to symbols.

2.4 Huffman coding

A. Steps to follow to implement algorithm

- **Step 1: Count the symbols's frequencies**
  - Read input and count the frequencies of each symbol in the data.
  - The frequencies will show how often each symbol appears in the input data.
- **Step 2: Build a Min-Heap (priority queue)**
  - Insert each symbol as a node into Min-heap based on its frequency.
  - The highest priority will be the least frequency symbols.
- **Step 3: Construct the Huffman tree**
  - Get the two lowest-frequency nodes and then remove them from the heap.
  - Merge them into a node and let the frequency of that node be the sum of the 2 frequencies of those 2 symbols.
  - Insert the newly created node back into the heap.
  - Repeat this step until there is one node in the heap. That node will be the root of the Huffman tree.
- **Step 4: Assign binary code to symbols**
  - Use the traverse algorithm to traverse the Huffman tree from the root and assign binary code to each symbol followed by this rule:
    - When passing the left edge, '0' will be added to the binary string
    - When passing the left edge, '1' will be added to the binary string
    - When meeting a leaf, the binary string will be used to assign the symbol in that node.
  - Based on the implementation of step 3, all symbols are leaf nodes. Because of that, no symbol is in the path of another symbol from the root so the binary code is prefix-code.
- **Step 5: Encode the data**
  - Replace all symbols with the Huffman code that had been assigned to them.
- **Step 6: Decode the data**
  - Using the Huffman tree to find the symbols assigned by the code in the encoding file.
  - Start from the root and read bit by bit in the encoding file:

    * When meet bit ‘1’, go to the right edge
    * When meet bit ‘0’, go to the left edge
    * When meeting a leaf, print the symbol in that node.
    * Return to root.
- Repeat until the end of the file.

3 Algorithm Theory

3.1 Implementation detail

A. Data structure

- **Huffman tree node structure**

```
struct HuffmanTreeNode
{
    int Freq = 0;
    char Character = 0;
    HuffmanTreeNode *Left = nullptr, *Right = nullptr;
    HuffmanTreeNode();

    HuffmanTreeNode(int Freq, char Character);
    HuffmanTreeNode(int Freq, HuffmanTreeNode *left, HuffmanTreeNode *right);
};
```

- Construct a Huffman tree node to store:
  - **Frequency (Freq):** sum of its child if it's an internal node or frequency of a symbol if it's a leaf.
  - **Symbol (Character):** store the symbols if it is a leaf node else default is 0.
  - **Its child (Left, Right):** store the pointer point to left and right child node address.
- There are 3 constructors:
  - **HuffmanTreeNode():** to create a default node with 0 frequency, 0 character, and no child (null pointer).
  - **HuffmanTreeNode(int Freq, char Character):** to create a leaf node with given frequency and the symbols it contains.
  - **HuffmanTreeNode(int Freq, HuffmanTreeNode *left, HuffmanTreeNode *right):** to create an internal node with the given frequency (usually sum of its child) and assign the address of its child to left and right.

- **Priority Queue (Min-Heap):**

```
struct PriorityQueue
{
    HuffmanTreeNode **NodeArray;
    int NumberOfElement, CurrentNumberOfElement = 0;
    PriorityQueue() = default;
    PriorityQueue(int n);

    ~PriorityQueue();

    void Init(int n);

    void sift(int l, int r);

    void Push(HuffmanTreeNode *x);

    HuffmanTreeNode *Pop();

    HuffmanTreeNode *Top();
};
```

  - Construct a priority queue for inserting nodes and getting the highest priority node (the smallest frequency node) to construct the Huffman tree.
    * **NodeArray:** stores the pointer to the Huffman Node.
    * **NumOfElement:** capacity (size) of the priority queue.
    * **CurrentNumberOfElement:** number of Nodes in the queue.
  - There are 3 constructors:
    * One for default priority queue
    * **PriorityQueue(int n):** Create a priority queue with capacity = n.
    * ∼ **PriorityQueue():** deconstruct the Queue (delete the allocated memory in the NodeArray).
  - Implement some functions for queue feature:
    * **Init():** Initialize the priority queue with capacity n.
    * **Sift(int l, int r):** perform sift down (min heap) to maintain the heap property.
    * **Pop():** to get the highest priority node and pop it out of the queue, then use sift to construct the root of the heap.
    * **Top():** to get the highest priority node.

```cpp
void PriorityQueue::Push(HuffmanTreeNode *x)
{
    NodeArray[CurrentNumberOfElement++] = x;
    int NewNumberIndex = CurrentNumberOfElement - 1;
    while (NewNumberIndex >= 0 && NodeArray[(NewNumberIndex - 1) /
    ↪ 2]->Freq > NodeArray[NewNumberIndex]->Freq)
    {
        std::swap(NodeArray[NewNumberIndex], NodeArray[(NewNumberIndex -
        ↪ 1) / 2]);
        NewNumberIndex = (NewNumberIndex - 1) / 2;
    }
}
```

* **Push(HuffmanTreeNode *x):** to add new element into the queue
  · Add the element x to the end of queue
  · Increase the current element by one.
  · Get the index of x and compare it to its parent ((NewnumberIndex - 1)/2). If x frequency is smaller, swap x with its parent.
  · Repeat the previous progress until x frequency larger than its parent or x is the root.

- **Label**
  - An array to store the binary code for characters to encode.
- **Trie node structure (for the decoding progress)**

```cpp
struct TrieNode
{
    char Character = 0;
    TrieNode *Left, *Right;

    TrieNode();
    TrieNode(char Character);
    TrieNode(TrieNode *left, TrieNode *right);
};
```

  - To decode Huffman code, the algorithm need to build a tree similar to the Huffman tree in the encode process.
  - Trie Node has a character variable to store symbols if it is a leaf node and two childs left and right.
  - There are 3 constructors, similar to Huffman tree.

B. Encoding progress:

- **Compute the frequency of each character:**

```cpp
while (FileToEncode.get(Character))
    Freq[Character]++;
```

  - Open a stream to input data (file), then read characters individually and increase frequency.

- **Build Huffman tree:**

```cpp
for (int i = 0; i < MAX_N; i++)
{
    if (Freq[i])
    {
        HuffmanTreeNode *NewNode = new HuffmanTreeNode(Freq[i], (char)i);
        PQ.Push(NewNode);
    }
}
```

  - Browse through each symbol, check if it appears in the input data then allocate a memory of **HuffmanTreeNode** to store its frequency and symbol.
  - Push the newly created node into the priority queue.

```cpp
while (PQ.CurrentNumberOfElement >= 2)
{
    HuffmanTreeNode *leftNode = PQ.Pop(), *rightNode = PQ.Pop();
    if (!leftNode || !rightNode)
    {
        std::cerr << "Error: Priority queue became empty." << std::endl;
        delete leftNode;
        delete rightNode;
        return;
    }
    HuffmanTreeNode *NewNode;
    NewNode = new HuffmanTreeNode(leftNode->Freq + rightNode->Freq,
    ↪  leftNode, rightNode);
}
```

  - Get the first and the second smallest frequency node out of the priority queue. Then, check if valid Node (not null Node)
  - Create an internal node which have frequency is the sum of them and its left child is the smaller frequency node, the rest is its right node.
  - Push that internal Node back into the priority queue
  - Repeat the progress until there is just one Node in the queue. That node will be the root of the Huffman tree.

- **Assign binary code to each symbol:**

```cpp
void HuffmanEncoding::Labeling(std::string *LabelArray,
                               HuffmanTreeNode *CurrentNode, std::string Label)
{
    if (!CurrentNode)
        return;
    if (!CurrentNode->Left && !CurrentNode->Right)
    {
        FileSize += Freq[CurrentNode->Character] * Label.size();
        LabelArray[CurrentNode->Character] = Label;
        return;
    }
    if (CurrentNode->Left)
        Labeling(LabelArray, CurrentNode->Left, Label + '0');
    if (CurrentNode->Right)
        Labeling(LabelArray, CurrentNode->Right, Label + '1');
}
```

  – Use the Depth First Search traversal algorithm to travel the Huffman tree.
  – Check if the current node is not a null node.
  – If the current node is a leaf node, then calculate the **FileSize += code length * frequency of that symbol**. Assign binary code (Label) to the ASCII value of the symbols position in the **LabelArray**.
  – If the current node is the internal node, go to its left child and add '0' to the code. Then, go to its right child and add '1' to the code.

- **Encode file:**

```cpp
int EncodedPointer = 0;
Buffer = 0;
NumberOfBit = 0;
size_t bufferSize = (FileSize == 0) ? 1 : (FileSize / 8) + ((FileSize % 8) !=
↪  0);
if (Encoded != nullptr)
    delete[] Encoded;
Encoded = new char[bufferSize];
```

  – Calculate the buffer size to write binary code in the **Encoded** char array to file.
  – Allocated memory for Encoded array based on the bufferSize.

```
while (FileToEncode.get(Character))
{
    for (char Bit : Label[Character])
    {
        Buffer <<= 1;
        if (Bit == '1')
            Buffer |= 1;
        NumberOfBit++;
        if (NumberOfBit == 8)
        {
            Encoded[EncodedPointer++] = Buffer;
            Buffer = 0;
            NumberOfBit = 0;
        }
    }
}
```

- Read symbols in the input file one by one, take its code and put it in the buffer.
- When enough 8 bits are added to the buffer (size of char) then put the buffer into the **Encoded** array.

```
EncodedFile.write((char *)&FileSize, sizeof FileSize);

size_t bytesToWrite = (FileSize == 0) ? 0 : (FileSize / 8) + ((FileSize % 8)
↪  != 0);

if (Encoded && bytesToWrite > 0)
{
    EncodedFile.write(Encoded, bytesToWrite);
}
```

- Write the file size and the **Encoded** array to the encoded file.

- **Write the Huffman code to file:**

```cpp
void HuffmanEncoding::WriteHuffmanCode(std::string HuffmanCodeFileName)
{
    std::ofstream HuffmanCode(HuffmanCodeFileName);
    if (!HuffmanCode.good())
    {
        std::cerr   << "Error: Could not open Huffman code output file: " <<
        ↪   HuffmanCodeFileName << std::endl;
        return;
    }
    if (!IsLabeled)
    {
        std::cerr   << "Error: Cannot write Huffman codes, labeling not
        ↪   complete." << std::endl;
        HuffmanCode.close();
        return;
    }
    for (int i = 0; i < MAX_N; i++)
    {
        if (Freq[i] > 0)
        {
            HuffmanCode << i << ' ' << Label[i] << '\n';
        }
    }

    HuffmanCode.close();
}
```

  - To serve the decode progress, the algorithm needs to store each symbol with its Huffman code.
  - For each distinct symbol, write the ASCII value and its Huffman code to another file.

C. Decoding progress:

- **Build Trie:**

```cpp
void DecodeHuffman::ReadHuffmanCode(std::string HuffmanCodeFileName)
{
    std::ifstream HuffmanCode(HuffmanCodeFileName);
    if (!HuffmanCode.good())
    {
        std::cerr << "Error: Could not open Huffman code file: " <<
        ↪  HuffmanCodeFileName << std::endl;
        IsHuffmanTreeGood = false;
        return;
    }
    while (HuffmanCode >> CharacterCode)
    {
        HuffmanCode >> Binary;
        TrieObject.Add(Binary, (char)CharacterCode);
    }
    IsHuffmanTreeGood = true;
    HuffmanCode.close();
}
```

  - Open a stream to file which contains ASCII values of symbols and their Huffman code.
  - Read the input data (symbols and code) and add it to the trie.

```cpp
void Trie::Add(std::string Binary, char Character)
{
    if (Head == nullptr)
        Head = new TrieNode();

    TrieNode *CurrentNode = Head;
    for (int i = 0; i < Binary.size(); i++)
    {
        if (Binary[i] == '1')
        {
            if (CurrentNode->Right == nullptr)
                CurrentNode->Right = new TrieNode();
            CurrentNode = CurrentNode->Right;
        }
        else
        {
            if (CurrentNode->Left == nullptr)
                CurrentNode->Left = new TrieNode();
            CurrentNode = CurrentNode->Left;
        }
    }
    CurrentNode->Character = Character;
}
```

- Check if trie does not contain any node, allocate root node memory for it.
- Start from the root and read the binary code; if bit is ‘1’, then go to the right node; if bit is ‘0’ go to the left node (if the child does not exist , allocate new memory for it).
- When reading the last bit, store the ASCII value of the symbol at the current node (leaf node).

• **Decode file:**

```
while (EncryptedFile.read(&Character, 1))
{
    for (int i = 7; i >= 0; i--
    {
        if (FileSize <= 0)
            break;
        if (Character & (1 << i))
            CurrentNode = CurrentNode->Right;
        else
            CurrentNode = CurrentNode->Left;

        if (!CurrentNode)
        {
            std::cerr << "Error: Decoding error - encountered invalid path in
            ↪  Huffman Trie." << std::endl;
            EncryptedFile.close();
            IsInputValid = false;
            return;
        }
        if (isLeaf(CurrentNode))
        {
            DecodedString.push_back(CurrentNode->Character);
            CurrentNode = TrieObject.Head;
        }
        --FileSize;
    }
}
```

- Open output stream to **encrypted** file.
- Read each 8 bytes in the file store to **Character**. Browse the binary code in **Character** from left to right to travel trie from the root. If bit is ‘0’ go to the left child and vice versa.
- When meeting a leaf node, push the symbol in that node to **DecodedString** and assign the current node by the root.
- Repeat the progress until the algorithm reaches the end of file.
- Then, write the **DecodedString** to the encrypted file.

3.2 Complexity:

A. Time complexity:

- L is the number of symbols in the input data.
- N is the number of distinct symbols.
- **Build Huffman tree:**
  - **Frequency Calculation:**
    - Browse each symbol in the input data to count its frequency: $O(L)$.
    - Total cost: $O(L)$.
  - **Priority Queue (Min-Heap):**
    - Use Min-heap structure to find the highest priority (smallest frequency) element: $O(logN)$.
    - Add all nodes into the queue: $O(N)$.
    - After adding each node, the priority queue has to swap that node into the correct position according to heap structure: $O(NlogN)$.
    - Total cost: $O(NlogN)$
  - **Build Huffman tree:**
    - Get the first smallest frequency node in the priority queue: $O(1)$.
    - Pop out the root and find a new root of heap structure in queue: $O(logN)$.
    - Get the second smallest frequency node and pop out it: $O(logN)$.
    - Create an internal node to connect 2 nodes just taken out: $O(1)$.
    - Add the internal node into the queue: $O(logN)$.
    - Get node 2 node until there is only one node in the queue take n - 1 times: $O(N)$.
    - Total cost: $O(NlogN)$.
  - **Average time complexity to build a Huffman tree:**
    - $O(L) + O(NlogN) + O(NlogN) = O(L + NlogN)$.
- **Encoding progress:**
  - Build a Huffman tree of the input data: $O(L + NlogN)$.
  - Traverse the Huffman tree to assign a code to each symbol.
    - Meet all internal nodes one time: $O(N)$.
    - Meet all leaf one time: $O(N)$.
    - Store code for each symbol: $O(N)$
    - Total cost: $O(3N) = O(N)$.
  - Replacing symbols with Huffman Codes:
    - Look up for a code of symbol: $O(1)$
    - Iterating through the input data: $O(L)$
    - Replace each symbol in input data with code have length C: $O(LC)$.
    - Total cost: $O(LC)$.
  - **The best case complexity:** The code length of each symbol is $logN$, so $C = logN$.
    - $O(L + NlogN) + O(N) + O(LlogN) = O(L + LlogN)$.
  - **Worst case:** The frequency of symbols form similar to a Fibonacci sequence.

        * The Huffman coding greedy always combines the two smallest frequency nodes. Since they are similar to Fibonacci numbers, the sum of two smallest frequencies will always be smaller than all node frequency.
        * The Huffman tree is not balanced, it seems like a linked list rather than a balanced binary tree.
        * The longest length of the Huffman code is N, so $C = N$.
        * **Time complexity:** $O(L + NlogN) + O(N) + O(LN) = O(L + LN)$

- **Decoding progress:**
    - Read the symbols with their codes in frequency file: $O(NC)$.
    - Build trie (similar to Huffman tree):
        * With each symbol, create a path on trie base on its binary code length C : $O(NC)$.
    - Replace Huffman codes with symbols
        * Read each bit in the encoded input data: $O(LC)$.
        * Traverse the trie to find symbol of each code: $O(LC)$.
        * Replace the code with symbol: $O(L)$.
        * Total cost: $O(2LC) + O(L) = O(LC)$.
    - **The best case complexity:** The code length of each symbol is $logN$, so $C = logN$.
        * $O(NlogN) + O(NlogN) + O(LlogN) = O(LlogN)$
    - **Worst case:** The frequency of symbols form similar to a Fibonacci sequence.
        * $O(N^2) + O(N^2) + O(LN) = O(LN)$.

B. Space complexity:

- **Huffman code:**
    - An array to store the frequency of symbols: O(N).
    - Huffman tree:
        * N - 1 internal nodes: $O(N - 1)$.
        * N leaf nodes: $O(N)$.
        * Total cost: $O(2N - 1) = O(N)$.
    - An array to store code of each symbol: $O(NC)$.
    - Total cost: $O(N) + O(N) + (NC) = O(NC)$.
- **Encode progress and Decode progress:**
    - Huffman code: $O(NC)$.
    - Encoded string of input data: $O(LC)$.
    - **The best case complexity:** The code length of each symbol is $logN$, so $C = logN$.
        * $O(NlogN) + O(LlogN) = O(LlogN)$
    - **Worst case:** The frequency of symbols form a Fibonacci sequence.
        * $O(N^2) + O(LN) = O(LN)$.

3.3 Mathematically optimal in Huffman coding:

A. Definition of entropy:

- In information theory, the entropy of a random variable quantifies the average level of uncertainty or information associated with the variable's potential states or possible outcomes.[7]
- In Huffman Coding, Entropy is the minimum average number of bits needed per symbol.
- In information theory, the self-information is a basic quantity derived from the probability of a particular event(data, bits) occurring from a random variable.[8]

$$I(x) := -\log_b[\Pr(x)] = -\log_b(P).$$

- Let source X have n symbols $S = \{s_1, s_2, \ldots, s_n\}$ with probabilities $P = \{p_1, p_2, \ldots, p_n\}$, the entropy $H(X)$ is given by:

$$H(X) = -\sum_{a=1}^{n} p_a \log_2 p_a$$

  - p_a is the probability of symbol s_a.
  - Using the logarithm base 2 ensures that entropy is measured in bits.

B. Entropy and Huffman Coding Efficiency

- If the algorithm encodes symbol a in the input string A with a binary code of length l_a, the expected length for encoding one symbol is:

$$L = \sum_{a \in A} p_a l_a,$$

  - p_a is the probability of symbols a
  - l_a is the length of binary code for symbols a.
- To prove that Huffman coding algorithm is a near-optimal encoding, we need to prove that the value of L is always close to the value of H(X). This mean proves that:

$$H(X) \leq L < H(X) + 1.$$

- Since the length of a symbol must be an integer and the minimum length of a symbol is the self-information of symbol.

$$l_a = \lceil -\log_2 p_a \rceil$$

- Which mean:

$$-\log_2 p_a \leq l_a < -\log_2 p_a + 1$$

- Multiply 3 expression with sum of p_a

$$\sum p_a(-\log_2 p_a) \leq \sum p_a l_i < \sum p_a(-\log_2 p_a + 1)$$

$$\Leftrightarrow -\sum p_a(\log_2 p_a) \leq L < -\sum p_a \log_2 p_a + \sum p_a$$

$$\Leftrightarrow H(X) \leq L < H(X) + 1$$

4 Algorithm Visualization

4.1 Huffman Algorithm Steps

A. Count the Symbols' Frequencies

- Read the input and count the frequency of each symbol in the data.
- The frequencies indicate how often each symbol appears in the input data.

Symbol	Frequency
H	1
u	1
m	1
a	1
n	1
f	2

Table 3: All symbols and their frequencies in "Huffman"

B. Build a Min-Heap (Priority Queue)

- Insert each symbol as a node into a Min-heap based on its frequency.
- The highest priority is given to the symbols with the least frequency.

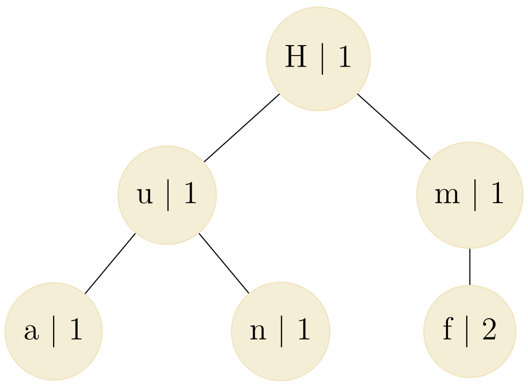


Figure 1: Step 1: Build a Min-Heap (Priority Queue)

C. Construct the Huffman Tree

- Extract the two nodes with the lowest frequency from the heap.
- Merge them into a new node, with the frequency of this node being the sum of the two frequencies.
- Insert the newly created node back into the heap.
- Repeat this step until there is only one node left in the heap. This node will be the root of the Huffman tree.

D. Assign Binary Codes to Symbols

- Use a traversal algorithm to traverse the Huffman tree from the root and assign binary codes to each symbol following these rules:

  - When passing the left edge, add '0' to the binary string.
  - When passing the right edge, add '1' to the binary string.
  - When reaching a leaf, use the binary string to assign the symbol in that node.
- All symbols are leaf nodes, ensuring that no symbol is in the path of another symbol from the root, making the binary code a prefix code.

Symbol	Huffman Code
H	000
u	001
f	11
m	010
a	011
n	10

Table 4: Huffman codes for "Huffman"

E. Encode the Data

- Replace all symbols with the Huffman codes that have been assigned to them.
- For the string "Huffman", the encoded bit sequence is:

Symbol	Code
H	000
u	001
f	11
f	11
m	010
a	011
n	10

Complete encoded string: 000 001 11 11 010 011 10

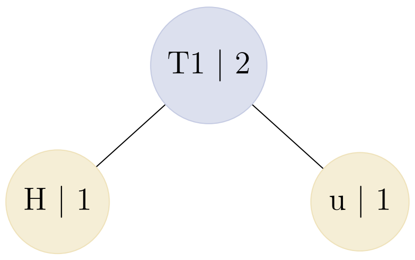
F. Decode the Data

- Use the Huffman tree to find the symbols assigned by the codes in the encoded file.
- Start from the root and read the encoded file bit by bit:
  - When encountering bit '1', go to the right edge.
  - When encountering bit '0', go to the left edge.
  - When reaching a leaf, print the symbol in that node.
- Return to the root and repeat until the end of the file is reached.
- Decoding process:

Code	Symbol	Progress
000	H	H
001	u	Hu
11	f	Huf
11	f	Huff
010	m	Huffm
011	a	Huffma
10	n	Huffman

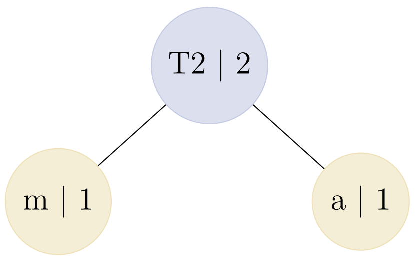
4.2 Huffman Tree Construction for "Huffman"

The following figures illustrate key steps in the construction of the Huffman tree for the string "Huffman":



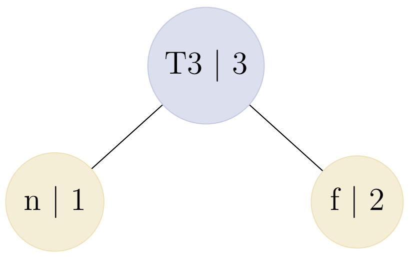
Symbol	Frequency
m	1
a	1
n	1
T1	2
f	2

Figure 2: Step 2: First Merge - H and u



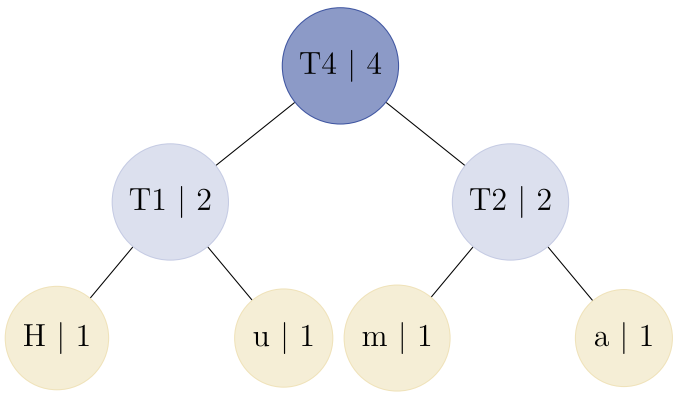
Symbol	Frequency
n	1
T1	2
T2	2
f	2

Figure 3: Step 3: Second Merge - m and a



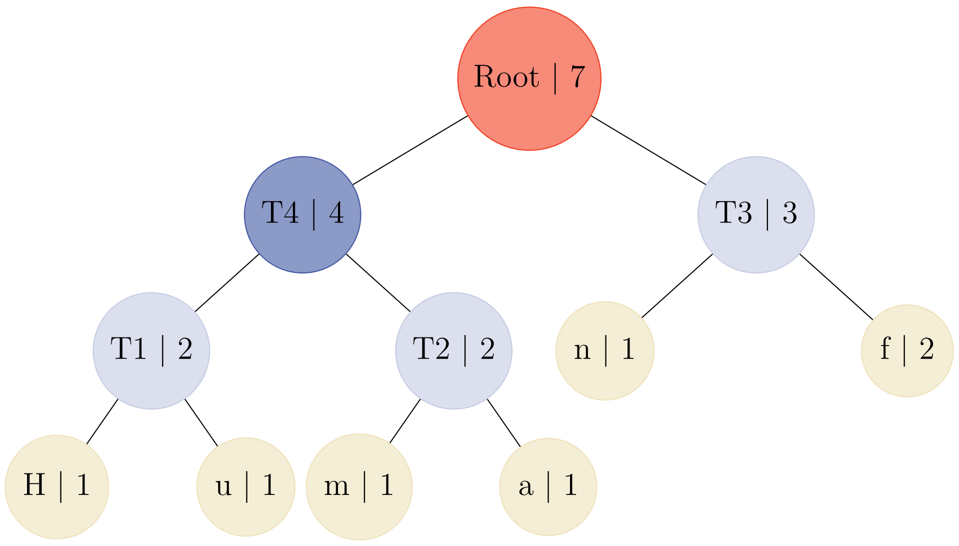
Symbol	Frequency
T1	2
T2	2
T3	3

Figure 4: Step 4: Third Merge - n and f



Symbol	Frequency
T3	3
T4	4

Figure 5: Step 5: Fourth Merge - T1 and T2



Symbol	Frequency
Root	7

Figure 6: Step 6: Final Merge - T3 and T4

4.3 Huffman Codes

Following the tree construction, we traverse it from the root and assign 0 to left child and 1 to the right child. This gives us the Huffman codes shown below:

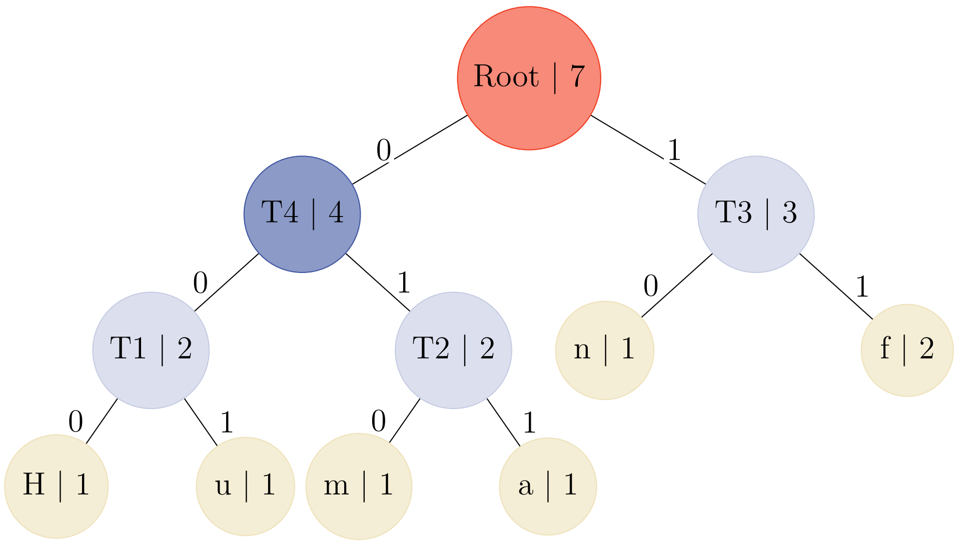


Figure 7: Final Huffman tree with edge labels

Symbol	Huffman Code	ASCII (for comparison)
H	000	01001000 (8 bits)
u	001	01110101 (8 bits)
f	11	01100110 (8 bits)
m	010	01101101 (8 bits)
a	011	01100001 (8 bits)
n	10	01101110 (8 bits)

Figure 8: All symbols and their Huffman codes for "Huffman"

4.4 Encoding Efficiency Analysis

- Original string "Huffman" (7 characters):
  - ASCII encoding: $7 \times 8 = 56$ bits
  - Huffman encoding: 18 bits (000 001 11 11 010 011 10)
  - Compression ratio: $(1 - 18/56) \times 100\% \approx 67.86\%$ savings.
- The Huffman encoding provides significant compression by assigning shorter codes to more frequent symbols.
- In this case, the character 'f' appears twice and receives the shortest code (11), while other characters receive longer codes.

5 Adaptive Huffman Coding

- Unlike static Huffman coding, where the code tree is built based on the given data set, which can not be changed while encoding. The adaptive version can construct and update the tree dynamically and is very useful when the frequency of the symbol is unknown or changes during transmission. There are 2 common algorithm variations to construct the dynamic Huffman tree.
- For the simplicity of this section, let's suppose that N embodies the length of the input data, M stands for the size of the character set and the algorithm can not have full access to the input data (streaming, video, data is being split into many chunks).

5.1 Faller-Gallager-Knuth algorithm (FGK)

FGK is known as one of the earliest adaptive Huffman coding algorithms.

A. Principles of FGK Huffman Coding

- The algorithm constructs the dynamic Huffman tree that evolves by reading symbols in the input data one by one. The tree starts with an empty node, indicating that no symbol is added to the tree. The Huffman tree grows as more symbols are encountered.
- There are some main principles:
  - **Initial empty tree:** The Huffman tree just includes a special node called the "0-node".
  - **Incremental updates:** increase frequency of a symbol when the symbol exists in the tree or create a new leaf to store that symbol.
  - **Tree rebalancing:** Swap nodes or branches of the tree if necessary to construct a Huffman tree and get the optimal compression.

B. The algorithm

- **Initialization:**
  - Create an empty tree with a 0-node.
  - Create a structure of a node to store its left (right) child, frequency, symbol, and node number (its position in a level-order traversal and from left to right at each level).
- **Encoding the symbol:**
  - For each symbol, if the symbol is already present in the Huffman tree, Encoder will send the binary code of that symbol. When a new symbol is encountered, Encoder will send the binary code of the 0-node and then send the ascii code of that symbol.
- **Processing symbols:**
  - After sending the code, the algorithm will process the symbol:
    - ∗ If it's a new symbol:
      - Take the stored position or traversal to the 0-node.
      - Replace the 0-node with a new parent node and its childs are
        - · Left node: for the new 0-node.

      · Right node: for the new symbols.
    - Update and rebalance the Huffman tree.

- **Updating and rebalancing the Huffman tree**
  - Locating the node: In the previous step, the algorithm already stands at the node that contains the symbol that was just processed.
  - Use a traversal algorithm to travel from the root and find a successor node that has a frequency equal to the current node (make sure that it is closer to the root).
  - Check if it is a valid successor node (not a parent of current node, not current node, and not a null node). If valid, swap successor and current node.
  - Increase the frequency of the current node by one.
  - Assign the current node to its parent.
  - Continue to find and swap until the current node reaches the root.
  - Increase the frequency of the root by one.
- **Decoding:**
  - Decoder will receive a binary code from the Encoder. If the binary code is 0-node then the next 8 bits will be the ascii code for the new symbol.
  - Encoder and Decoder both use the same algorithm so they will achieve the same tree for each bit sent and received.

A. Analyze the algorithm:

- **Time Complexity**
  - O(NM): For each symbol, the algorithm will traverse from root to leaf of the Huffman tree (in average case the Huffman tree height is log M, in the worst case scenario the Huffman tree height can be M).
- **Space Complexity:**
  - O(M): Huffman tree requires about M nodes to store each symbol.

5.2 Vitter algorithm (Algorithm V)

A. Introduction

- Vitter's algorithm is an improvement of the FGK algorithm. To help make Huffman Coding more consistent. Basic principles of the V algorithm are the same as in the FGK algorithm.
- The Vitter algorithm has important terminologies and constraints:
  - **Implicit Numbering:** Nodes are numbered in increasing order from left to right and from bottom to top. This means we have to set the left node with a lower value, the higher value to the right node, and a higher value than the right node to their parent node.
  - **Blocks:** formed by Nodes that have the same weight and the same type (leaf node or internal node).
  - **Leader:** highest numbered node in a block.

B. Algorithm

- Like the FGK algorithm, Vitter starts with an empty tree with a "NTY" node (Not Yet Transmission), which is similar to the 0-node. Encoding and processing symbols with the same method of FGK.
- The difference comes from updating and rebalancing the Huffman tree process.
- Instead of traversing the Huffman tree to find a successor which has the same frequency with current node then swap them and increase the frequency until reach to the root. The Vitter's algorithm finds the leader of the current node's block and uses a Sliding Up Mechanism to slide and swap the current node into the right position, then increase the frequency and repeat this progress until it reaches the root.
- Step-by-Step process:
  - **Increment**
    - Increase the frequency of the current node by one.
  - **Determine the Weight Block**
    - Find the block that the current node belongs to based on its frequency.
  - **Find the leader of that block**
    - Find the Node that has the highest order in that block.
  - **Swap (if necessary)**
    - If the current node is already the leader, there is no need to swap.
    - Otherwise, swap the current node and the leader.
  - **Move to the Parent**
    - Slide the current node to its parent.

C. The improvement

- **Faster tree update:** The FGK algorithm might need to swap nodes multiple times, but for V algorithm by carefully managing blocks of nodes and an implicit numbering scheme, Vitter's algorithm guarantees that the tree update operation takes amortized constant time (O(1)). Some updates might take longer to run, but on average, updates over a long sequence of operations are constant. This leads to significantly faster overall encoding and decoding.
- **More efficient tree restructuring:** FGK algorithm when restoring the sibling property after increment often involves a series of swapping operations. On the other hand, Vitter's update procedure is designed to minimize node movements.

D. The trade offs

- **Increased complexity:** The primary trade-off is that the algorithm is more complex and hard to understand and implement correctly compared to FGK.
- **Similar Compression:** Both FKG and V algorithms have a generally similar compression ratio. Neither of them provides a significantly better performance than the other.

E. Complexity

- **Time complexity:** O(N log M) because every update(occurs on every symbol) needs to traverse from the root to the symbol leaf (Huffman tree height on average is about log M).
- **Space complexity:** O(M) since the algorithm needs to store the Huffman tree.

F. Conclusion

- Vitter's algorithm built upon FGK's principles and provided a mathematically proven faster method to update the Huffman tree, achieving amortized constant runtime that makes it helpful for big data sets with large character sets. This speed comes with a cost of increasingly difficult implementation.

5.3 Similarities and dissimilarities

A. Similarities

- **Core principle:** Both Static and Adaptive Huffman Coding algorithms are fundamentally similar since they are both based on the Huffman tree.
- **Frequency-based compression:** Both algorithms assign shorter codes to more frequent symbols and longer codes to less frequent symbols
- **Prefix-free code:** Both generate prefix codes (prefix-free codes), meaning no codeword is a prefix of another codeword.
- **Variable-length code:** Unlike fixed-length schemes, both algorithms assign codes of different bit lengths. They share the strategy of minimizing the average code length by allocating shorter binary sequences to symbols that occur more often and longer sequences to less common symbols.
- **Decoding mechanic:** Although the trees are different(one static, one dynamic), the core decoding process for both involves traversing the Huffman tree from the root based on the incoming bits until a leaf node is reached.

B. Dissimilarities

- **Single pass processing:**
  - For Static Huffman Coding, the algorithm needs two passes over the data (read the data twice): one to count frequency to build the Huffman tree and a second pass to encode using that Huffman tree.
  - For Adaptive Huffman Coding(FGK and Vitter's algorithm) only needs a single pass over data to learn the frequency and update the Huffman tree as it processes the data. This is a necessity as in some scenarios the algorithm can read the data once (data stream).
- **Adaptability:**
  - Static Huffman Coding uses a single fixed Huffman tree based on the overall frequency of the data. If this tree is used on different data inputs with a divergent character frequency, the output code will be suboptimal for later data input.
  - The Huffman tree of FGK and V algorithm is dynamic and adaptive to the local frequency of the input data, as the character frequency changes the algorithm will reflect the change to the Huffman tree, which leads to better overall compression.
- **Reduced overhead:**
  - Static Huffman Coding needs to store and transmit the frequency table or the Huffman tree so the Decoder can reconstruct the Huffman tree. This adds a significant overhead for the algorithm, especially for the small data input.

  - For the Adaptive Huffman Coding, the Decoder will build the same adaptive tree, following the exact same update rule as the Encoder. Therefore, there is no need to explicitly store or transmit Huffman trees, eliminating this overhead.
- **Suitabilities for complex data scheme:**
  - For Static Huffman Coding, the algorithm generally needs to know the input data beforehand to calculate the frequency, making it less suitable for steaming applications where data arrives sequentially and the total size might be unknown.
  - The Adaptive Huffman Coding algorithm operates in a single pass and on the fly, so that it's suitable for compressing data streams in real time application.

6 Other variants of Huffman coding

6.1 n-ary Huffman Coding:

- The n-ary Huffman coding is a version of the algorithm that uses the numbers from 0 to n - 1 to encode symbols and build a tree. Similar to the original version to build a Huffman tree, except that a node now has maximum n child and the n least frequency symbols are taken together instead of 2.
- This variant is useful in the following scenarios:
  - **High-speed data transmission:** With the higher code character, the Huffman tree depth is decreased, so it optimizes numbers of steps to encoding and decoding data.
  - **Large Alphabet size:** when dealing with a large symbol set like DNA sequence, etc, this approach can reduce the complexity to the original Huffman coding.

6.2 Length-Limited Huffman Coding:

- This variant can limit the maximum length of the code assigned to a symbol and ensure that no symbol has excessively long code. The package-merge algorithm used to make this variant more efficient can provide the content, which is the maximum length of the codeword.
- This variant is useful in the following scenarios:
  - **Memory-Constrained Environments:** This variant can reduce memory to store the code of a symbol, which can help when working on a low-RAM system.
  - **Low-Latency Decoding:** Length-limited Huffman coding ensures a predictable worst-case decoding time when used for real-time application.

6.3 Canonical Huffman Coding:

- Canonical Huffman Coding is a refinement of the standard Huffman coding algorithm designed primarily to reduce the overhead associated with storing or transmitting the Huffman tree structure itself. While standard Huffman coding generates an optimal set of prefix codes given symbol frequencies, the representation of the tree needed for decoding can consume significant space, diminishing the overall compression effectiveness, especially for files with large alphabets or relatively short lengths. Canonical Huffman Coding achieves the same compression optimality (in terms of average code length) as standard Huffman but represents the codebook using a more compact, standardized method based solely on the lengths of the codewords.
- The ideal scenario for this variant:
  - **Large alphabet size:** Compressing data with a wide range of possible symbols, such as 16-bit Unicode characters, large dictionaries in LZ77/LZW compression outputs, or quantized coefficients in image/audio compression (JPEG/MP3).
  - **Frequent Transmission of Small Data Blocks:** Compressing many small, independent messages or data packets, perhaps in a network protocol or for storing small records.

7 Performance Experiment

7.1 Overview

Huffman Encoding optimizes storage by assigning shorter bit representations to more frequently occurring characters in the provided text data. While Huffman Encoding can be applied to any binary file, this test focuses specifically on ASCII text for simplicity. The test data is generated using predefined character sets to analyze how different character distributions impact compression efficiency. The code used to generate these test files is provided for reproducibility.

7.2 Testing procedure

- **Test File Generation**: generate the test file using a predefined character set.
- **Huffman Encoding and Decoding**: the generated test file is encoded using Huffman Encoding algorithm and then decoded back to its original form.
- **Verification**: compare the input file and the decoded version to ensure the correctness of the implemented Huffman Encoding algorithm.
- **Data Collection**: saving the size of the input file, encoded version for further analysis.

7.3 Dataset Description

- The dataset consists of four different input sizes: 500, 2000, 100000, and 10000000 characters.
- The generated text files are based on four predefined character sets:
  - "**Lower**": Only lowercase letters and whitespace (28 unique characters).
  - "**LowerUpper**": Includes both lowercase and uppercase letters (54 unique characters).
  - "**ASCII**": Contains lowercase, uppercase, digits and whitespace (64 unique characters).
  - "**Printable**": Includes all printable ASCII characters (97 unique characters).

7.4 Result

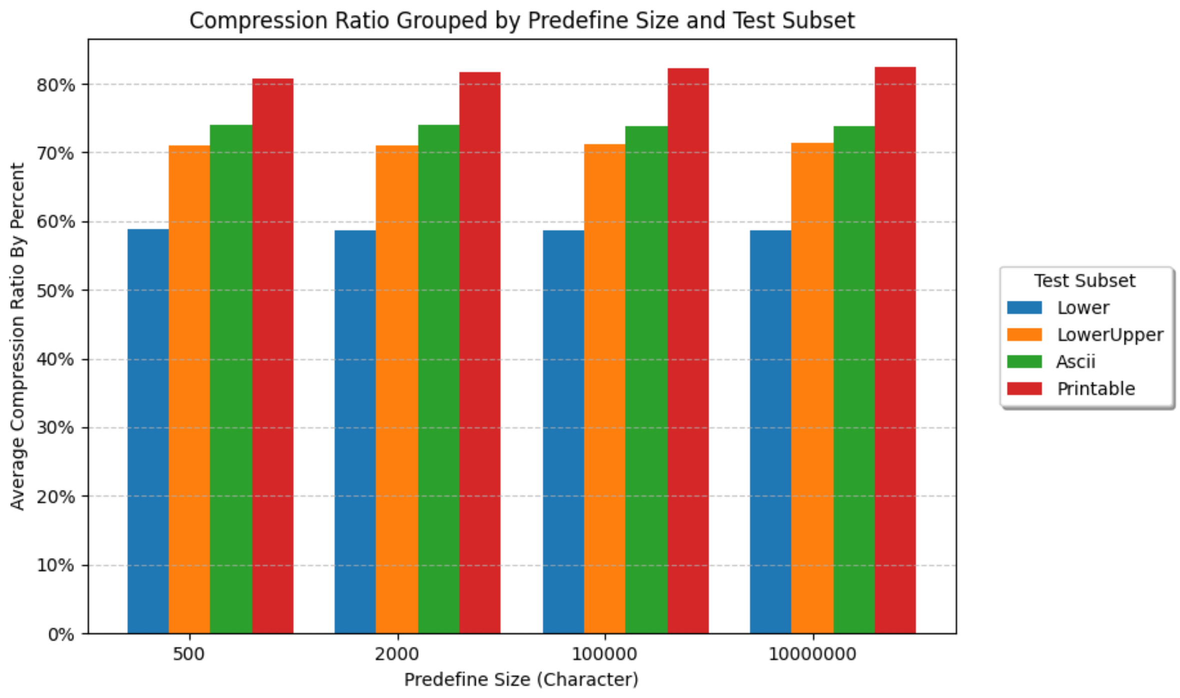


Figure 9: Compression Ratio Group by File size and Character set

- This graph is presented as part of the experimental evaluation of a Huffman coding implementation. It visually summarizes the compression performance achieved across different input file sizes and character sets. The y-axis represents the "**Average Compression Ratio By Percent**" which we'll define precisely below. The x-axis groups the results by "**Predefined Size by Character**" representing the total number of characters in the input files. Within each size group, there are four bars, each representing a different "**Test Subset**" (character set).
- The compression ratio is calculated using the formula:

$$CompressionRatio = (EncodedFileSize/OriginalFileSize) * 100\%$$

  A lower compression ratio is better, indicating a greater reduction in file size. A ratio of 100% means that there are no compression; a ratio of 50% indicate that the encoded file is half the size of the original. A ratio above 100% would imply that the "**compressed**" file is actually larger than the original (which is possible with Huffman coding if the input data has a very uniform distribution of characters).
- The graph shows four distinct file size groups: 500, 2000, 100000 and 10000000 characters. Within each character set, the compression ratio remains consistent across these different file sizes. This suggests the effectiveness of Huffman Coding is largely **independent of overall file size**.
- **The strong dependence of the compression ratio on the character set is directly due to the structure of the Huffman tree**. Larger character sets, particularly those with many low-frequency characters, result in taller trees and longer average code lengths, decreasing the compression efficiency.

7.5 Conclusion

These results confirm that Huffman Coding is an effective lossless compression technique for ASCII text files. Under standard conditions (Printable character set), Huffman Encoding reduces file size by approximately 20%. In specific cases, such as datasets with highly repetitive characters, the reduction can be as high as 30%.

8 Advantage and disadvantage of the Static Huffman Coding

8.1 Advantages

- Huffman Coding using the probability of the symbol, helps produce an optimal prefix code. The code generated is guaranteed to be the lowest average code length for this specific distribution.
- Huffman Coding is a lossless encoding technique, guaranteeing the data is intact after decode.
- Huffman Coding has a relatively simple implementation.
- Huffman Coding is widely used and supported (JPEG, MP3, DEFLATE)
- Decoding a Huffman encoded code is generally very fast since it only needs to traverse the Huffman Tree.

8.2 Disadvantages

- Require a prior knowledge of the data (frequencies) to build Huffman Tree to encode input.
- Huffman Coding is not adapted, if the frequencies slightly differ from the start the compression ratio will be suboptimal.
- The overhead of storing the Huffman Tree is also a problem.
- Huffman Coding treats each symbol independently, so it can't take advantage of correlations between symbols ("q" is almost always followed by "u").
- Can be inefficient for super large character sets as already shown in the IO section.
- Vulnerable to Bits error. A single bit change can resolve the whole decoded text being wrong since Huffman Coding does not have an error correction part.

9 Discover the Application of Algorithm

Huffman coding is a fundamental of lossless data compression so it is applied in various fields such as File Compression, Multimedia Compression, Data Transmission and Networking, Artificial Intelligence and Machine Learning, Hardware and Embedded Systems.

A. File Compression

- Huffman coding is a common algorithm used in file compression because it helps to reduce file size without losing information.
- For example:
  - 7-Zip: a popular open-source program for archiving files. It uses Deflate compression which is a combination of LZ77 and Huffman coding.
  - WinZip and WinRAR: These are commercial file archiving tools that also use DEFLATE compression, including Huffman coding.
  - macOS Archive Utility (built-in .zip functionality): The built-in archive utility in macOS creates and extracts ZIP files and it relies on DEFLATE.
  - Command-line tools (gzip, zip): common tools found in Unix-like systems (Linux, macOS), their primary compression method is Deflate.

B. Multimedia Compression

- Huffman coding is also used in compressing images, audio, and video. The algorithm can reduce file sizes while maintaining quality.
- For example:
  - JPEG (Image Compression): It uses Huffman coding to compress pixel data after Discrete Cosine Transform (DCT).
  - MP3 (Audio Compression): Huffman coding is used to compress quantized frequency data. It helps to reduce redundancy in audio streams.
  - MPEG-4 (Video Compression): Huffman coding is used to compress motion vector data in video frames.

C. Data Transmission and Networking

- By reducing the amount of data sent over the internet, Huffman coding helps to improve network efficiency.
- For example:
  - HTTP Content-Encoding (GZIP): Web servers use GZIP which includes Huffman coding to compress CSS, HTML, and Javascript files and speed up web loading times.
  - Fax Machines and Telephony: They use Group 3 fax encoding which includes Huffman coding to compress text-based data for faster transmission.

D. Artificial Intelligence and Machine Learning

- Huffman coding is applied in these fields for efficient memory usage.
- For example:
  - Decision Tree Compression: optimize large decision trees in machine learning.

  - Natural Language Processing (NLP): Huffman coding is applied in dictionary-based text compression for language modeling.

E. Hardware and Embedded Systems

- To compress real-time data, hardware chips, and embedded devices need to implement Huffman coding.
- For example:
  - GPUs and FPGAs: real-time compression in high-performance computing (HPC).
  - Data Storage Systems: reduce file size in SSDs and HDDs.

10 Compare with other Compression Algorithms

10.1 Introduce

Huffman Coding is a solid foundation for data compression. While being fundamental, its performance and suitability must be evaluated against other lossless compression algorithms to have a clear picture of the world of compression algorithms. This section provides a comprehensive comparison between Huffman Coding and other significant lossless compression algorithms, especially RLE (Run-Length Encoding), Arithmetic Coding, and the Lempel-Ziv (LZ) family of methods. This evaluation will primarily focus on key performance metrics, including compression efficiency, computational complexity (encoding/decoding speeds), adaptability to varying data characteristics, and associated overhead.
Let's assume that N represents the length of the input data and M speaks for the size of the character set.

10.2 Huffman Coding vs RLE

Feature	**Huffman Coding**	**Run-Length Encoding(RLE)**
Principle	Variable-length codes based on symbol frequencies.	Replaces repeated substrings with references.
Complexity	Time: O(N + M log M). Space: O(M).	Time: O(N). Space: O(N).
Adaptivity	Static version is not adaptive; adaptive versions exist but are more complex.	Not inherently adaptive.
Overhead	Requires storing the Huffman tree (or frequencies) to decode.	Minimal overhead (just the counts).
Use Cases	General-purpose compression, JPEG, MP3, GZIP.	Simple images (e.g., fax), data with long runs.
Strengths	Good overall compression, widely supported.	Very fast and simple for suitable data.
Weaknesses	Can't take advantage of correlations betweens symbols.	Poor compression for data without long runs.

Table 5: Huffman Coding vs RLE

10.3 Huffman Coding vs Arithmetic Coding

Feature	Huffman Coding	Arithmetic Coding
Principle	Variable-length codes based on symbol frequencies.	Encodes the entire message as a single fraction.
Complexity	Time: O(N + M log M). Space: O(M).	Time: O(N + M). Space: O(M).
Adaptivity	Static version is not adaptive; adaptive versions exist but are more complex.	Can be easily made adaptive.
Overhead	Requires storing the Huffman tree (or frequencies) to decode.	Minimal overhead (typically just model parameters).
Use Cases	General-purpose compression, JPEG, MP3, GZIP.	JPEG 2000, JBIG2, high-performance compression.
Strengths	Good overall compression, widely supported.	Better compression, especially for small alphabets.
Weaknesses	Can't take advantage of correlations between symbols.	More complex, slower encoding/decoding.

Table 6: Huffman Coding vs Arithmetic Coding

10.4 Huffman Coding vs Lempel-Ziv Algorithms (LZ77, LZ78, LZW)

Feature	Huffman Coding	Lempel-Ziv Algorithms
Principle	Variable-length codes based on symbol frequencies.	Replaces repeated substrings with references.
Complexity	Time: O(N + M log M). Space: O(M).	Time: O(N). Space: O(N).
Adaptivity	Static version is not adaptive; adaptive versions exist but are more complex.	Adaptive, learns patterns in the data.
Overhead	Requires storing the Huffman tree (or frequencies) to decode.	Minimal overhead (dictionary is built dynamically).
Use Cases	General-purpose compression, JPEG, MP3, GZIP.	GZIP, ZIP, PNG (LZ77 + Huffman), GIF (LZW).
Strengths	Good overall compression, widely supported.	Good for data with repeating patterns, adaptive.
Weaknesses	Can't take advantage of correlations between symbols.	Can be slower than Huffman.

Table 7: Huffman Coding vs Lempel-Ziv Algorithms

10.5 Huffman Coding vs Burrows-Wheeler Transform (BWT)

Feature	**Huffman Coding**	**Burrows-Wheeler Transform**
Principle	Variable-length codes based on symbol frequencies.	Reorders data to group similar characters together. Not a compression algorithm itself, but a preprocessor.
Complexity	Time: O(N + M log M). Space: O(M).	Time: O(N log N) Space: O(N).
Adaptivity	Static version is not adaptive; adaptive versions exist but are more complex.	Not applicable.
Overhead	Requires storing the Huffman tree (or frequencies) to decode.	Minimal overhead, needs to store the index of the original string.
Use Cases	General-purpose compression, JPEG, MP3, GZIP.	bzip2
Strengths	Good overall compression, widely supported.	Very high compression ratio, especially for text.
Weaknesses	Can't take advantage of correlations between symbols.	Relatively slow.

Table 8: Huffman Coding vs Burrows-Wheeler Transform

10.6 Conclusion

There are no perfect compression algorithms, the performance of these compression algorithms is heavily dependent on the input data and specific requirements (compression ratio, speed, memory usage, etc.). Huffman Coding is a versatile and widely used technique, often used with other algorithms to create a more complex compression scheme (JPEG, GZIP, DEFLATE).

References

[1] GeeksforGeeks. *Huffman Coding | Greedy Algorithm.* Accessed: 2025-03-19. 2024. URL: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

[2] Solomon Golomb. "Run-length encodings (corresp.)". In: *IEEE transactions on information theory* 12.3 (1966), pp. 399–401.

[3] Giovanni Manzini. "The Burrows-Wheeler Transform: Theory and Practice: Invited Lecture". In: *International Symposium on Mathematical Foundations of Computer Science.* Springer. 1999, pp. 34–47.

[4] Jorma Rissanen and Glen G Langdon. "Arithmetic coding". In: *IBM Journal of research and development* 23.2 (1979), pp. 149–162.

[5] David Salomon and Giovanni Motta. *Handbook of Data Compression.* 5th. Springer, 2010. ISBN: 978-1-84882-902-2. DOI: 10.1007/978-1-84882-903-9.

[6] Peter Shor. *Huffman Coding.* Accessed: 2025-03-29. n.d. URL: https://math.mit.edu/~shor/18.310/huffman.pdf.

[7] Wikipedia contributors. *Entropy (information theory) — Wikipedia, The Free Encyclopedia.* Accessed: 2025-03-29. 2025. URL: https://en.wikipedia.org/wiki/Entropy_(information_theory)#Definition.

[8] Wikipedia contributors. *Information content — Wikipedia, The Free Encyclopedia.* Accessed: 2025-03-29. 2025. URL: https://en.wikipedia.org/wiki/Information_content.

[9] Jacob Ziv and Abraham Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE transactions on Information Theory* 24.5 (1978), pp. 530–536.