國立臺南大學資訊工程學系 112 學年度第一學期

演算法
第四次作業
報告名稱：Huffman Coding 壓縮

班級：資工三
學號：S11059006
姓名：謝昊君

日期：113 年 1 月 2 日

目錄

(一) 簡介及問題描述

1. 簡介

給定一個 PGM image 檔案，利用 Huffman Coding 演算法設計與實作檔案壓縮程式：

i. Input: xxx.pgm (using IrfanView (https://www.irfanview.com/) to covert jpeg, gif, and more into pgm ASCII format)
ii. Output: a compressed file xxx.hc
iii. Functions:

a. 壓縮(Compression) <= 須說明如何將 Huffman Tree and Codes 置於檔案中

b. 解壓縮(Uncompression)

c. 多個檔案壓縮成一個檔案，可個別加入壓縮與解壓縮 (optional: 加分用)

2. 問題

This is a summary of the main steps to be executed upon choosing this option:

i. Read the specified image (ASCII format) and count the frequency of all pixels in the image.

a) D:\> Huffman-Coding_Prog_學號 -c test.pgm

ii. Create the Huffman coding tree using a Priority Queue based on the pixel frequencies.
iii. Create the table of encodings for each pixel from the Huffman coding tree.
iv. Encode the image and output the encoded/compressed image.
v. Read the encoded/compressed file you just created, decode it and output the decoded image.

a) D:\> Huffman-Coding_Prog_學號 -d test.hc testd.pgm

Your program must provide the following required output:

i. the pixel frequencies (i.e., image histogram)
ii. the table of pixel values and their bit encoding (sorted from smallest to largest)
iii. original image size and compressed image size
iv. compressed image and uncompressed image

(二) 理論分析

簡介

Huffman code 是一種廣泛應用於數據壓縮的算法，由大衛·霍夫曼於 1952 年提出。它是一種基於字符頻率的變長編碼方法，目的是將常用的字符用更短的位元組（bits）表示，而不常用的字符則用較長的位元組表示，從而實現數據的有效壓縮。

原理

1. 頻率分析：算法首先對待壓縮數據（例如圖像檔案）進行頻率分析，統計每個字符（或像素值）出現的頻率。

2. 構建霍夫曼樹：

- 創建一個葉節點森林，每個節點代表一個字符及其頻率。
- 將這些節 點根據頻率排序後，放入優先隊列（最小堆）。
- 每次從隊列中取出兩個頻率最小的節點，創建一個新節點，其頻率是這兩個節點頻率之和，並將這兩個節點作為新節點的子節點。
- 重複此過程，直到隊列中只剩下一個節點，該節點即為霍夫曼樹的根節點。

3. 生成編碼表：

- 從霍夫曼樹的根節點開始，向下遍歷至每個葉節點。
- 每當向左子節點移動時，添加一個“0”到編碼中；向右子節點移動時，則添加一個“1”。
- 當到達葉節點時，形成的二進制序列即為該節點字符的 Huffman code。

效率

Huffman code 的效率取決於輸入數據的特性。對於頻繁出現的字符（或像素值）分配較短的編碼，可以大幅減少數據的整體大小。這種方法尤其適用於具有不均勻分布的字符集的數據壓縮。

(三) 演算法則

步驟 1: 頻率統計

- 從輸入文件（例如 PGM 圖像）中讀取每個像素值。
- 統計每個不同像素值出現的頻率。
- 將這些像素值和相應的頻率存儲在一個映射結構中，用於後續步驟。
- 時間複雜度：這一步驟涉及遍歷整個圖像一次，以統計每個像素值的頻率。因此，時間複雜度與圖像中像素的總數成線性關係，即 O(n)，其中 n 是圖像中像素的數量。
- 空間複雜度：需要一個數據結構（如散列表）來存儲每個唯一像素值及其對應的出現次數。在最壞情況下，每個像素值都是唯一的，因此空間複雜度為 O(u)，其中 u 是唯一像素值的數量。

步驟 2: 構建霍夫曼樹

- 為映射結構中的每個像素值創建一個節點，並將這些節點放入一個優先隊列（最小堆）。
- 每次從隊列中取出兩個最小頻率的節點，創建一個新節點作為它們的父節點，其頻率是兩個子節點頻率之和。
- 將新節點重新加入優先隊列。
- 重複此過程，直到隊列中只剩下一個節點，該節點成為霍夫曼樹的根節點。
- 時間複雜度：這一步驟涉及將節點插入和從優先隊列中移除，每次操作的時間複雜度為 O(logk)，其中 k 是隊列中的元素數量。由於每次插入和移除操作都會減少隊列中的元素數，整體時間複雜度為 O(ulogu)。
- 空間複雜度：構建霍夫曼樹需要存儲所有節點，因此空間複雜度為 O(u)。

步驟 3: 生成編碼表

- 從霍夫曼樹的根節點開始，進行深度優先遍歷。
- 向左子節點移動時添加“0”，向右子節點移動時添加“1”。
- 到達葉節點時，生成的二進制序列即為該像素值的 Huffman code。
- 將所有葉節點的像素值及其對應的編碼存儲在一個映射表中。
- 時間複雜度：生成編碼表需要對霍夫曼樹進行一次深度優先遍歷，時間複雜度為 O(u)。對圖像進行編碼則需要遍歷每個像素，時間複雜度為 O(n)。
- 空間複雜度：編碼表的空間複雜度為 O(u)，而編碼後的圖像大小取決於編碼長度，通常小於或等於原始圖像的大小。

步驟 4: 圖像編碼

- 使用生成的編碼表將原始圖像數據（像素值）轉換為 Huffman code。

- 逐個像素讀取圖像數據，並用對應的編碼替換。
- 結果是一個由二進制編碼組成的字符串，代表壓縮後的圖像數據。

步驟 5: 數據和樹結構序列化

- 將霍夫曼樹的結構轉換為一個可以存儲在文件中的序列化格式。
- 將序列化的樹結構和編碼後的圖像數據一起保存到壓縮文件中。
- 時間複雜度：序列化霍夫曼樹的時間複雜度為 **O(u)**。
- 空間複雜度：序列化結構所需的空間取決於樹的大小，大致為 **O(u)**。

步驟 6: 解壓縮

- 從壓縮文件中讀取並反序列化霍夫曼樹。
- 使用反序列化的霍夫曼樹解碼壓縮的圖像數據。
- 將解碼後的像素值還原為原始圖像格式。
- 時間複雜度：解壓縮包括兩部分，一是重建霍夫曼樹，二是使用該樹解碼圖像數據。重建樹的時間複雜度為 **O(u)**，解碼操作的時間複雜度為 **O(n)**
- 空間複雜度：同樣需要 **O(u)** 的空間來存儲霍夫曼樹，以及額外的空間來存儲解碼後的圖像數據。

(四) 程式設計環境架構

程式設計語言、工具、環境與電腦硬體等規格說明...

1. 程式語言

C++ in MS Windows

2. 程式開發工具

Visual C++ 2022

3. 電腦硬體

CPU: AMD R7

Main Memory: 16GB

(五) 程式 (含 source code, input code, and output code)

1. 主程式

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <unordered_map>
#include <map>
#include <queue>
#include <iomanip>
#include<chrono>
//auto start = std::chrono::high_resolution_clock::now();
//auto end = std::chrono::high_resolution_clock::now();
//auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
//cout << "Running time of compression" << duration.count() << " microseconds." <<
```

```
endl;
struct Node {
	int value; // Pixel value
	int freq;    // Frequency of the pixel
	Node* left;
	Node* right;

	Node() : value(-1), left(nullptr), right(nullptr) {}
	Node(int value, int freq) : value(value), freq(freq), left(nullptr), right(nullptr) {}
	Node(int value) : value(value), freq(0), left(nullptr), right(nullptr) {}
	Node(Node* left, Node* right) : value(0), freq(0), left(left), right(right) {}

};

struct Compare {
	bool operator()(Node* a, Node* b) {
		return a->freq > b->freq; // Min Heap
	}
};

// Function to construct Huffman Tree by minHeap
Node* buildHuffmanTree(std::unordered_map<int, int>& frequencyMap) {
	std::priority_queue<Node*, std::vector<Node*>, Compare> minHeap;

	// Create nodes for each unique pixel and add to min heap
	for (const auto& pair : frequencyMap) {
		minHeap.push(new Node(pair.first, pair.second));//Sort after pushing into heap
	}

	// Build Huffman Tree
	while (minHeap.size() > 1) {
		Node* left = minHeap.top();//Least
		minHeap.pop();
		Node* right = minHeap.top();//Second least
		minHeap.pop();

		//建一個 new node 左右連起加進去
		Node* combined = new Node(-1, left->freq + right->freq); // -1 indicates an
internal node
		combined->left = left;
		combined->right = right;

		minHeap.push(combined);
	}

	return minHeap.top(); // Root of Huffman Tree
}

// Function to encode Huffman Tree
void encodeHuffmanTree(Node* root, std::string str, std::unordered_map<int,
```

```
std::string>& huffmanCode) {
	if (root == nullptr) {
		return;
	}

	// Found a leaf node
	if (!root->left && !root->right) {
		huffmanCode[root->value] = str;
	}

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

// Function to encode specified Image
std::string encodeImage(const std::vector<std::vector<int>>& image,
                                    const std::unordered_map<int, std::string>& huffmanCode)
```

```
{
	std::string encodedImage;
	for (const auto& row : image) {
		for (int pixel : row) {
			encodedImage += huffmanCode.at(pixel);
		}
	}
	return encodedImage;
}

// Function to turn string(encoded image) into binary
std::vector<char> stringToBinary(const std::string& encodedString) {
	std::vector<char> binaryData;
	char currentByte = 0;
	int bitCount = 0;

	for (char bit : encodedString) {
		// Set the bit in currentByte
		currentByte = (currentByte << 1) | (bit == '1');

		bitCount++;
		if (bitCount == 8) {
			// Byte is filled, add to vector
			binaryData.push_back(currentByte);
			currentByte = 0;
			bitCount = 0;
		}
	}

	// Handle the last byte if it's not full
	if (bitCount > 0) {
		currentByte <<= (8 - bitCount);
		binaryData.push_back(currentByte);
	}

	return binaryData;
}

// Function to serialize huffman tree for compressing
void serializeTree(Node* root, std::vector<char>& encoding) {
	if (root == nullptr) return;

	if (!root->left && !root->right) {
		encoding.push_back('1');
		encoding.push_back(root->value);
	}
	else {
		encoding.push_back('0');
		serializeTree(root->left, encoding);
		serializeTree(root->right, encoding);
	}
}
```

```
// Function to reconstruct the Huffman tree
Node* decodeHuffmanTree(std::ifstream& file) {
    if (file.eof()) return nullptr;

    char marker;
    file.get(marker);

    if (marker == '1') {
        if (file.eof()) return nullptr;
        char ch;
        file.get(ch);
        return new Node(static_cast<unsigned char>(ch));
    }
    else {
        Node* left = decodeHuffmanTree(file);
        Node* right = decodeHuffmanTree(file);
        return new Node(left, right);
    }
}

// Function to decode the encoded data using the Huffman tree
std::vector<int> decodeData(Node* root, const std::string& encodedData) {
    std::vector<int> decodedData;
    Node* current = root; // For decoding a part of bits

    // Tracing tree using VLR to decode only for leaves
    for (char byte : encodedData) {
        for (int i = 7; i >= 0; --i) {
            bool bit = (byte >> i) & 1; //  提取每個位元

            if (bit == 0)
                current = current->left;
            else
                current = current->right;

            if (current->left == nullptr && current->right == nullptr) {
                decodedData.push_back(current->value);
                current = root; //  重置到根節點
            }
        }
    }
    return decodedData;
}

// Function to draw the TABLE of pixel values and corresponding Huffman code
void printHuffmanTable(const std::unordered_map<int, std::string>& huffmanCode) {
    // Sort the map
    std::map<int, std::string> orderedHC(huffmanCode.begin(), huffmanCode.end());

    //  表格標題
    std::cout << std::endl;
```

```
    std::cout << std::left << std::setw(12) << "Pixel Value" << "Huffman Code" <<
std::endl;
    std::cout << std::string(30, '-') << std::endl; //  分隔線

    // 遍歷映射並打印每個項目
    for (const auto& pair : orderedHC) {
        std::cout << std::left << std::setw(12) << pair.first << pair.second << std::endl;
    }
    std::cout << std::endl;
}

// Function to draw HISTOGRAM of pixel values and corresponding frequency
void printHistogram(const std::unordered_map<int, int>& frequencyMap) {

    // 找出最大頻率
    int maxFrequency = 0;
    for (const auto& pair : frequencyMap) {
        if (pair.second > maxFrequency) {
            maxFrequency = pair.second;
        }
    }

    // 定義最大長度和正規化比例
    const int maxLength = 100; // 可以自定義最大長度
    double normalizationRatio = static_cast<double>(maxLength) / maxFrequency;

    std::map<int, int> orderedFmap(frequencyMap.begin(), frequencyMap.end());


    std::cout << "Histogram of frequency: (one # indicates " << 1 / normalizationRatio
```

```
<< " times, no # indicates less than the number)" << std::endl;
        for (const auto& pair : orderedFmap) {
                int pixelValue = pair.first;
                int normalizedLength = static_cast<int>(pair.second * normalizationRatio);

                std::cout << std::left << std::setw(4) << pixelValue    << ": ";
                for (int i = 0; i < normalizedLength; ++i) {
                        std::cout << "#";
                }
                std::cout << std::endl;
        }

}

std::streamsize getFileSize(const std::string& filePath) {
        std::ifstream file(filePath, std::ifstream::ate | std::ifstream::binary);

        if (!file) {
                return -1;
        }

        // Get size of file
        std::streamsize size = file.tellg();
        file.close();
        return size;
}

int main(int argc, char* argv[]) {
        if (argc < 3) {
                std::cerr << "Usage: " << argv[0] << " -c/-d filename" << std::endl;
                return 1;
        }
        std::string mode = argv[1];          // "-c" or "-d"
        std::string filename = argv[2]; // File name, e.g., "test.pgm" or "test.hc"
        std::string filename2;                  // File name, e.g., "tested.pgm"
        if (argc >= 4) filename2 = argv[3];

        // For encoding
        if (mode == "-c") {
                // Compression code
                std::cout << "Compressing " << filename << std::endl;


                auto start = std::chrono::high_resolution_clock::now();


                std::ifstream file(filename);
                if (!file.is_open()) {
                        std::cerr << "Error opening file" << std::endl;
                        return 1;
                }
```

```
std::string line;
getline(file, line);    // Read the magic number (e.g., "P2")
getline(file, line);    // Read the next line which might be a comment
if (line[0] == '#') {
	getline(file, line); // Read the next line if the previous was a comment
}

std::stringstream ss(line);
int width, height, maxVal;
ss >> width >> height;    // Read image dimensions
file >> maxVal;                // Read maximum pixel value

std::vector<std::vector<int>> pixels(height, std::vector<int>(width));
std::unordered_map<int, int> frequencyMap;//<index, frequency>
for (int i = 0; i < height; ++i) {
	for (int j = 0; j < width; ++j) {
		file >> pixels[i][j];
		++frequencyMap[pixels[i][j]];
	}
}

file.close();

// Now, pixels[][] contains the pixel values of the PGM image

// Build huffman tree
Node* root = buildHuffmanTree(frequencyMap);

// Build huffman code and Serialize
std::unordered_map<int, std::string> huffmanCode;//<index, encode number>
encodeHuffmanTree(root, "", huffmanCode);
std::vector<char> treeStructure;
serializeTree(root, treeStructure);

// Encode specified image
std::string encodedImageData = encodeImage(pixels, huffmanCode);
std::vector<char> binaryData = stringToBinary(encodedImageData);

// Open a file stream to write
std::string compressedFile = filename + ".hc";
std::ofstream outputFile(compressedFile, std::ios::binary);
if (outputFile.is_open()) {
	outputFile.write(reinterpret_cast<const char*>(&width), sizeof(width));
	outputFile.write(reinterpret_cast<const char*>(&height), sizeof(height));

	// Output char(byte) by char
	outputFile.write(treeStructure.data(), treeStructure.size());
	outputFile.write(binaryData.data(), binaryData.size());
	outputFile.close();

	auto end = std::chrono::high_resolution_clock::now();
	auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
```

```
- start);
                std::cout << "Running time of compression is " << duration.count() << "
milliseconds." << std::endl;

                std::cout << "Image successfully encoded and saved to '"<<
compressedFile <<"'" << std::endl;
            }
            else {
                std::cerr << "Unable to open file for writing." << std::endl;
                return 1;
            }

            printHuffmanTable(huffmanCode);// Print Table
            printHistogram(frequencyMap);    // Print Histogram

            std::streamsize originalSize, compressedSize;
            originalSize = getFileSize(filename);// Print Original size
            compressedSize = getFileSize(compressedFile);// Print Compressed size

            if (originalSize >= 0) {
                std::cout << "Original image size: " << originalSize << " bytes." <<
std::endl;
            }
            else {
                std::cerr << "Error reading file size." << std::endl;
            }
            if (compressedSize >= 0) {
                std::cout << "Compressed image size: " << compressedSize << " bytes."
<< std::endl;
            }
            else {
                std::cerr << "Error reading file size." << std::endl;
            }
            //std::cout << "Compression ratio: " << (originalSize - compressedSize) /
```

```
originalSize << " %" << std::endl;
        }
        // For decoding
        else if (mode == "-d") {
                // Decompression code
                std::cout << "Decompressing " << filename << std::endl;

                auto start = std::chrono::high_resolution_clock::now();

                std::ifstream inputFile(filename, std::ios::binary);
                if (!inputFile.is_open()) {
                        std::cerr << "Unable to open file." << std::endl;
                        return 1;
                }

                // read width and height
                int width, height;
                inputFile.read(reinterpret_cast<char*>(&width), sizeof(width));
                inputFile.read(reinterpret_cast<char*>(&height), sizeof(height));

                // read tree and reconstruct
                Node* root = decodeHuffmanTree(inputFile);

                // read rest of encoded data and decode
                std::string encodedData(std::istreambuf_iterator<char>(inputFile), {});
                std::vector<int> decodedImage = decodeData(root, encodedData);
                std::cout << "File .hc successfully inputed" << std::endl;
                inputFile.close();

                //------------output below---------------
                std::ofstream outputFile(filename2);
                if (outputFile.is_open()) {
                        outputFile << "P2" << std::endl
                                        << width << " " << height << std::endl
                                        << "255" << std::endl;

                        for (int value : decodedImage) {
                                outputFile << value << " " ;
                        }
                        auto end = std::chrono::high_resolution_clock::now();
                        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end
- start);
                        std::cout << "Running time of depression is " << duration.count() << "
milliseconds." << std::endl;

                        std::cout << "Image successfully decoded and saved to '" << filename2 <<
```

```
""" << std::endl;

            }
            else {
                std::cerr << "Unable to open file for writing." << std::endl;
                return 1;
            }


            outputFile.close();

        }
        else {
            std::cerr << "Invalid mode. Use -c for compression and -d for decompression."
<< std::endl;
            return 1;
        }



        return 0;
}
```

2. Input Code Format

Three of examples for input use are in below….

(1) std::stringstream ss(line);

(2) for (int i = 0; i < height; ++i) {

```
    for (int j = 0; j < width; ++j) {

        file >> pixels[i][j];

        ++frequencyMap[pixels[i][j]];

    }

}
```

(3) std::string encodedData(std::istreambuf_iterator<char>(inputFile), {});

3. Output Code Format

Three of examples for output use are in below….

(1)

void printHuffmanTable(const std::unordered_map<int, std::string>& huffmanCode) {

```
    // Sort the map
```

```
    std::map<int, std::string> orderedHC(huffmanCode.begin(), huffmanCode.end());

    // 表格標題
    std::cout << std::endl;
    std::cout << std::left << std::setw(12) << "Pixel Value" << "Huffman Code" << std::endl;
    std::cout << std::string(30, '-') << std::endl; // 分隔線

    // 遍歷映射並打印每個項目
    for (const auto& pair : orderedHC) {
        std::cout << std::left << std::setw(12) << pair.first << pair.second << std::endl;
    }
    std::cout << std::endl;
}
```

(2)

```
void printHistogram(const std::unordered_map<int, int>& frequencyMap) {

    // 找出最大頻率
    int maxFrequency = 0;
    for (const auto& pair : frequencyMap) {
        if (pair.second > maxFrequency) {
            maxFrequency = pair.second;
        }
    }

    // 定義最大長度和正規化比例
```

```
        const int maxLength = 100; //  可以自定義最大長度
        double normalizationRatio = static_cast<double>(maxLength) / maxFrequency;

        std::map<int, int> orderedFmap(frequencyMap.begin(), frequencyMap.end());

        std::cout << "Histogram of frequency: (one # indicates " << 1 / normalizationRatio << " times, no # indicates less than the number)" << std::endl;
        for (const auto& pair : orderedFmap) {
            int pixelValue = pair.first;
            int normalizedLength = static_cast<int>(pair.second * normalizationRatio);

            std::cout << std::left << std::setw(4) << pixelValue    << ": ";
            for (int i = 0; i < normalizedLength; ++i) {
                std::cout << "#";
            }
            std::cout << std::endl;
        }

    }
```

(3)

```
    std::streamsize getFileSize(const std::string& filePath) {
        std::ifstream file(filePath, std::ifstream::ate | std::ifstream::binary);

        if (!file) {
            return -1;
```

```
    }

    // Get size of file
    std::streamsize size = file.tellg();
    file.close();
    return size;
}

if (originalSize >= 0) {
    std::cout << "Original image size: " << originalSize << " bytes." << std::endl;
}
else {
    std::cerr << "Error reading file size." << std::endl;
}
if (compressedSize >= 0) {
    std::cout << "Compressed image size: " << compressedSize << " bytes." << std::endl;
}
else {
    std::cerr << "Error reading file size." << std::endl;
}
```

(六) 執行結果、討論與心得

執行結果與討論 (執行時間、problem *n* 的大小等問題討論)等…

1. 執行結果

i.

Compressing test.pgm
Image successfully encoded and saved to 'test.hc'

Pixel Value	Huffman Code
0	1111011110000000111
1	1111011110000000110
2	1111011110000000101
3	1111011110000000100
4	001011001111100110
5	111101111000000000
6	11110111101011001
7	001011001111100111
8	11110111101011000
9	111101111010111100
10	111101111010111101
11	11110111101011111
12	1111011110000111
13	00101100111110010
14	111101111000000001
15	0010110011111000
16	1111011110000010
17	001011001111000
18	001011001111101
19	1100100011000101
20	1111011110101110
21	1111011110000001
22	1111011110101101
23	1100100011000100
24	1111011110000011
25	1111011110000110
26	111101111000010
27	111101111010010
28	110010001100011
29	001011001111001
30	00000101110011
31	00101100111101
32	111101111010011
33	00101100111111
34	00000101110010
35	11001000110111
36	11001000110000
37	0000010111000

38	11001000110110
39	11110111101010
40	11110111101000
41	0000010111111
42	0000010111110
43	1100100011010
44	1100100011001
45	1111011110001
46	000001011110
47	000001011101
48	001011001110
49	111101111001
50	111101111011
51	00101100110
52	11001000111
53	0000010110
54	0010110010
55	1100100010
56	1111011111
57	000001010
58	010101110
59	011110000
60	110101111
61	111110101
62	00000011
63	00101101
64	01010110
65	01111101
66	01111111
67	11010110
68	11100011
69	11101011
70	11100110
71	11011000
72	11100100
73	11010010
74	11011010
75	11001110
76	11001100
77	01111100
78	01111001
79	01101010
80	01110011
81	01011011
82	01110000
83	01110001
84	01011110
85	01010001
86	01010000
87	00101010
88	00110110
89	00011011

90	00001111
91	00100100
92	00001001
93	00001010
94	00001100
95	00011000
96	00011010
97	00100101
98	00110010
99	00110100
100	01001010
101	01001100
102	00101011
103	01011000
104	01010100
105	01101111
106	01011111
107	01010011
108	00110011
109	00001011
110	00000010
111	111110001
112	111100100
113	111100111
114	111000001
115	110100010
116	110101110
117	110011110
118	110010011
119	011100101
120	011110100
121	110010010
122	011101100
123	011101101
124	010101010
125	011011101
126	011011100
127	010101011
128	010110101
129	011100100
130	011110101
131	011111100
132	110100111
133	110100001
134	110111001
135	111000000
136	111011001
137	111000100
138	111000101
139	110111000
140	110011111
141	110101000

142	111001011
143	111011000
144	111011010
145	111011101
146	111011110
147	111101110
148	111011100
149	111010100
150	111010101
151	110111100
152	110111101
153	111001110
154	110110010
155	110111010
156	111010010
157	110110110
158	110110111
159	111001111
160	111011111
161	111011011
162	111110011
163	111111010
164	111111000
165	111111110
166	111111001
167	111111111
168	111111011
169	111110100
170	111101011
171	111101010
172	111100101
173	110110011
174	110111011
175	111101000
176	111101001
177	111110010
178	00000100
179	00001000
180	00100011
181	00101000
182	00101110
183	00110101
184	00101111
185	00100110
186	00100010
187	00101001
188	01001110
189	01101100
190	01011001
191	01011101
192	01101000
193	01111011

194	01110101
195	01110111
196	01110100
197	01101101
198	11001101
199	11101000
200	11111011
201	0000011
202	0001111
203	0100100
204	0011000
205	0001110
206	11110110
207	11010101
208	00011001
209	111111101
210	110101001
211	011111101
212	010100100
213	010010111
214	001000011
215	001011000
216	000011011
217	001000000
218	000011010
219	001000001
220	010100101
221	010010110
222	010011110
223	010011111
224	010101111
225	010110100
226	011110001
227	110100011
228	110100000
229	111001010
230	111111100
231	00100111
232	01011100
233	01101011
234	01001101
235	00001110
236	111110000
237	111100110
238	111010011
239	110100110
240	110010000
241	011010010
242	001000010
243	011010011
244	11100001
245	10

246	11000
247	01000
248	01100
249	00010
250	00110111
251	00111
252	1111000
253	11011111
254	0000000
255	1100101

Histogram of frequency: (one # indicates 1169.04 times, no # indicates less than the number)

0 :
1 :
2 :
3 :
4 :
5 :
6 :
7 :
8 :
9 :
10 :
11 :
12 :
13 :
14 :
15 :
16 :
17 :
18 :
19 :
20 :
21 :
22 :
23 :
24 :
25 :
26 :
27 :
28 :
29 :
30 :
31 :
32 :
33 :
34 :
35 :
36 :
37 :
38 :
39 :

40 :
41 :
42 :
43 :
44 :
45 :
46 :
47 :
48 :
49 :
50 :
51 :
52 :
53 :
54 :
55 :
56 :
57 :
58 :
59 :
60 :
61 : #
62 : #
63 : #
64 : #
65 : #
66 : #
67 : #
68 : #
69 : #
70 : #
71 : #
72 : #
73 : #
74 : #
75 : #
76 : #
77 : #
78 : #
79 : #
80 : #
81 : #
82 : #
83 : #
84 : #
85 : #
86 : #
87 : #
88 : #
89 : #
90 : #
91 : #

92 : #
93 : #
94 : #
95 : #
96 : #
97 : #
98 : #
99 : #
100 : #
101 : #
102 : #
103 : #
104 : #
105 : #
106 : #
107 : #
108 : #
109 : #
110 : #
111 : #
112 :
113 :
114 :
115 :
116 :
117 :
118 :
119 :
120 :
121 :
122 :
123 :
124 :
125 :
126 :
127 :
128 :
129 :
130 :
131 :
132 :
133 :
134 :
135 :
136 :
137 :
138 :
139 :
140 :
141 :
142 :
143 :

144 :
145 :
146 :
147 :
148 :
149 :
150 :
151 :
152 :
153 :
154 :
155 :
156 :
157 :
158 :
159 :
160 :
161 :
162 : #
163 : #
164 : #
165 : #
166 : #
167 : #
168 : #
169 : #
170 :
171 :
172 :
173 :
174 :
175 :
176 :
177 : #
178 : #
179 : #
180 : #
181 : #
182 : #
183 : #
184 : #
185 : #
186 : #
187 : #
188 : #
189 : #
190 : #
191 : #
192 : #
193 : #
194 : #
195 : #

196 : #
197 : #
198 : #
199 : #
200 : ##
201 : ##
202 : ##
203 : ##
204 : ##
205 : ##
206 : #
207 : #
208 : #
209 : #
210 :
211 :
212 :
213 :
214 :
215 :
216 :
217 :
218 :
219 :
220 :
221 :
222 :
223 :
224 :
225 :
226 :
227 :
228 :
229 :
230 : #
231 : #
232 : #
233 : #
234 : #
235 : #
236 : #
237 :
238 :
239 :
240 :
241 :
242 :
243 :
244 : #
245 :
##

246 : ############
247 : ##########
248 : ############
249 : #########
250 : #
251 : ##########
252 : ###
253 : #
254 : ##
255 : ###
Original image size: 1767136 bytes.
Compressed image size: 352913 bytes.

ii.

Decompressing test.hc

File .hc successfully inputed

Image successfully decoded and saved to 'tested.pgm'

iii.

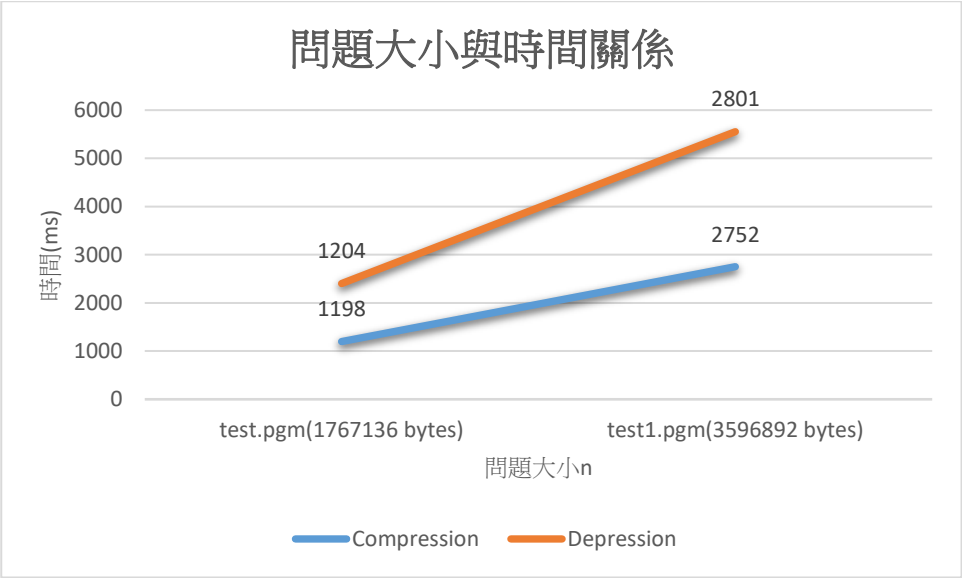
Invalid mode. Use -c for compression and -d for decompression.

2. 討論

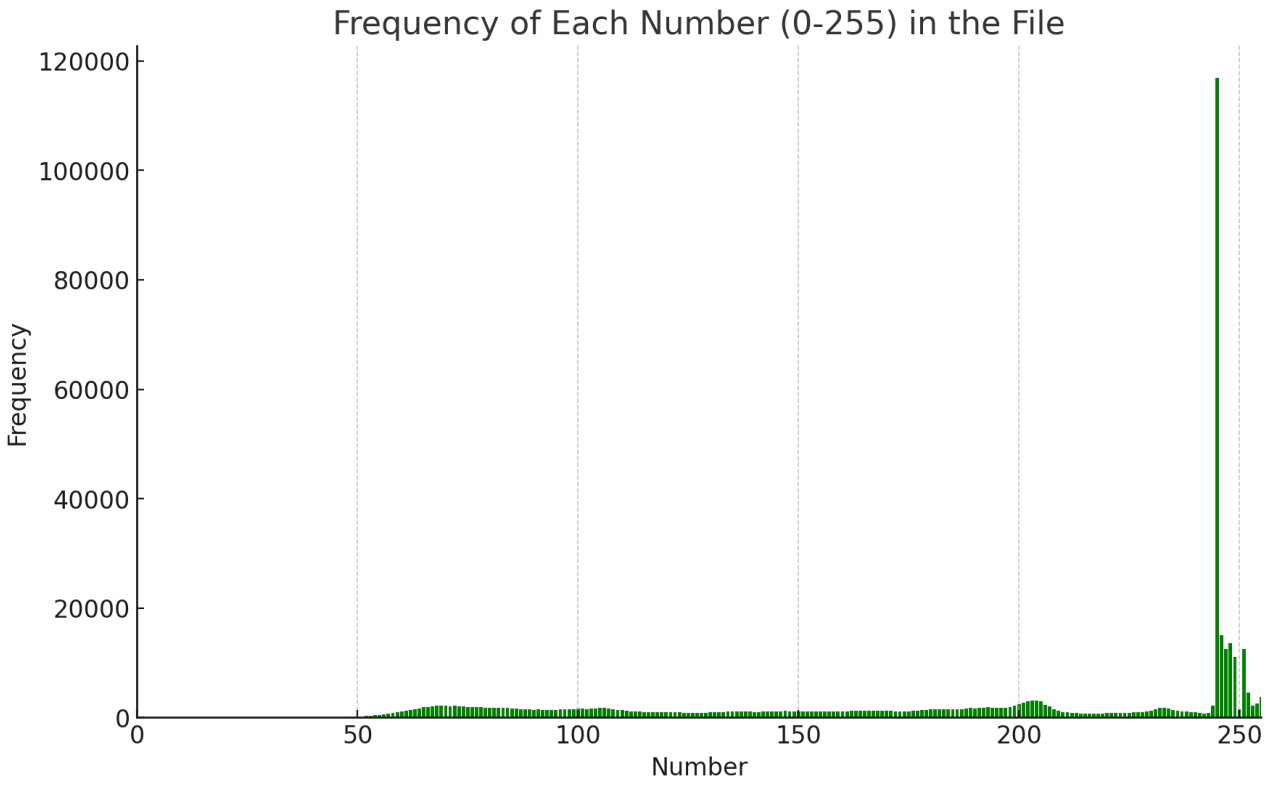
執行時間、問題大小等問題討論! 利用 MS Excel 畫出問題大小與執行時間的關係!

	Compression	Depression
test.pgm (1767136 bytes)	1198ms	1204ms
test1.pgm (3596892 bytes)	2752ms	2801ms

上表為 Running Time, Problem size *n 關係圖*



上圖為問題大小 n 與時間關係(ms)

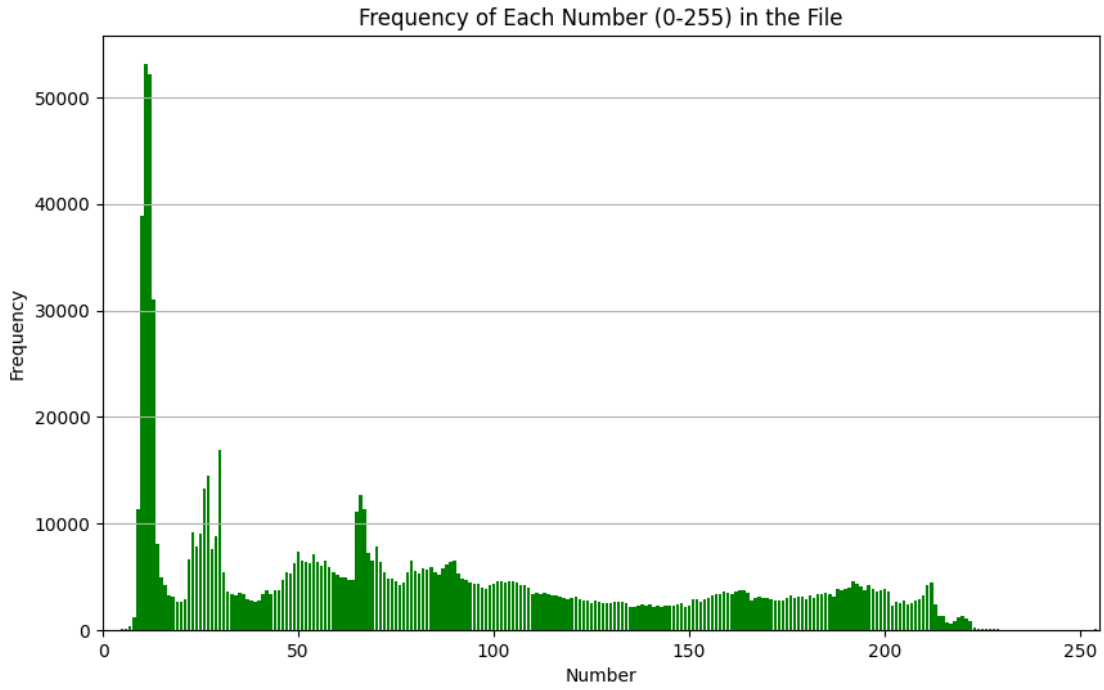


上圖為 test.pgm 的每個 pixel 對應的 frequency 長條圖

Original image size: 1767136 bytes.

Compressed image size: 352913 bytes.

Compression ratio: 80%



上圖為 test1.pgm 的每個 pixel 對應的 frequency 長條圖

Original image size: 3596892 bytes.

Compressed image size: 966895 bytes.

Compression ratio: 73%

由問題大小 n 與時間關係(ms)的關係圖可以看出問題愈大執行時間愈長

由上兩圖可看出分佈不平均的圖壓縮比可達到 70%以上，分佈越極端壓縮比越高。

3. 心得

學習體驗

在這次的演算法作業中，我深入研究了 Huffman code 的原理和實作。Huffman code 不僅是一個強大的數據壓縮工具，同時也是一個展現數據結構和算法設計巧妙結合的典範。學習過程中，我對於如何從基礎的數據結構（如 Priority queue 和 binary tree）出發，設計出一個高效的數據壓縮算法有了更深刻的理解。

實作挑戰

實作 Huffman code 時，我遇到了一些挑戰，特別是在構建 Huffman tree 和序列化/反序列化樹結構的過程中。這些步驟需要細致的考慮和精確的編程技巧。此外，確保 encoding 和 decoding 過程的高效性也是一大挑戰，這需要對算法進行細致的優化。

(七)參考文獻

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms," Third Edition, The MIT Press, 2009.

[2] R.C.T. Lee, S.S. Tseng, R.C. Chang, and Y.T.Tsai, "Introduction to the Design and Analysis of Algorithms," McGraw-Hill, 2005.

[3] Anany V. Levitin, "Introduction to the Design and Analysis of Algorithms," 3rd Edition, Addison Wesley, 2012.

[4] Richard Neapolitan and Kumarss Naimipour, "Foundations of Algorithms," Fourth Edition, Jones and Bartlett Publishers, 2010.

[5] …