SRM Institute of Science and Technology Department of

Department of Computer Science and Technology

Even Semester (2021-22)

Design Analysis and Algorithms

18CSC204J

CSE-I1

4th Semester

DAA Mini Project

Submitted by:

Name	Registration Number
Shreya Khera	RA2011003010582
Aditi Goel	RA2011003010599
Sidarth Bakshi	RA2011003010577

Submitted to :

Dr Meenakshi N.

Department of Computer Science and Engineering

CONTENT TABLE

Team Contributions:

Shreya Khera – Problem Identification, Real-life problem identifying, design of the algorithm.

Aditi Goel – Inputs needed for the algorithm, finding suitable algorithm for the problem, Time complexity.

Sidarth Bakshi – Writing code for the given algorithm, reference finding, and design of the algorithm

Problem Statement

Images are highly widespread in today's environment. The photos are saved in JPEG, GIF, PNG, and other formats. As a result, we cannot utilize the same picture size everywhere, thus we must compress and use it.

Problem Explanation:

The objective of image compression is to reduce the irrelevance and redundancy of the image data to be able to store or transmit data in an efficient form. It is concerned with minimising the number of bits required to represent an image. Image compression may be lossy or lossless.

For example, let us take a sample compressed and uncompressed image.

Uncompressed image:

Compressed image:

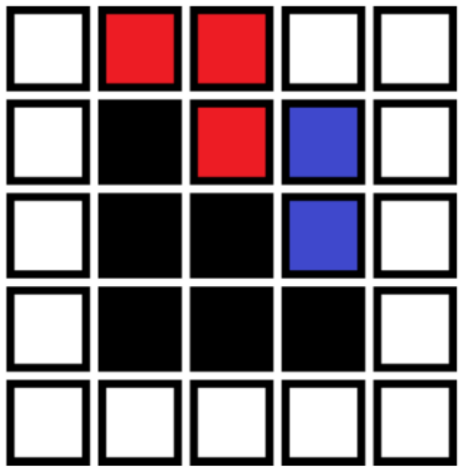
Although the both images look the same, we have compressed them using an algorithm. The algorithm we are going to use is Huffman coding.

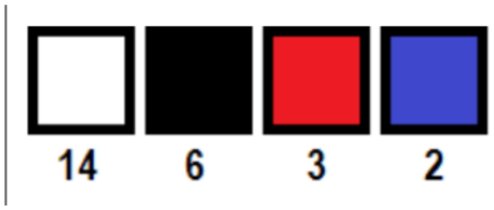
Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters; the lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

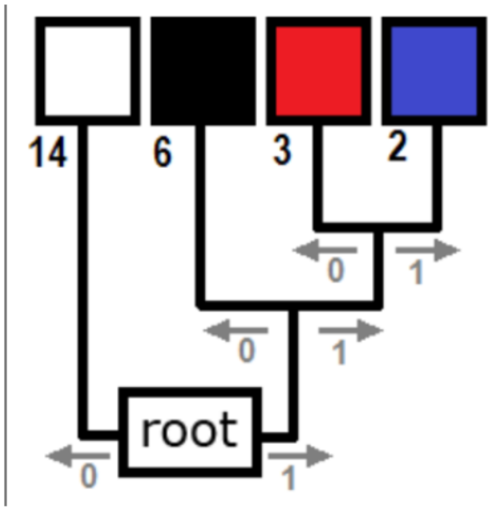
The variable-length codes assigned to input characters are Prefix Code, meaning the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Design Technique:

Huffman coding can be used to compress all sorts of data. It is an entropy-based algorithm that relies on an analysis of the frequency of symbols in an array. Huffman coding can be demonstrated most vividly by compressing a raster image. Suppose we have a 5×5 raster image with 8-bit colour, i.e. 256 different colours. The uncompressed image will take 5 x 5 x 8 = 200 bits of storage.







Our result is known as a Huffman tree. It can be used for encoding and decoding. Each colour is encoded as follows. We create codes by moving from the root of the tree to each colour. If we turn right at a node, we write a 1, and if we turn left – 0. This process yields a Huffman code table in which each symbol is assigned a bit code such that the most frequently occurring symbol has the shortest code, while the least common symbol is given the longest code.

Because each colour has a unique bit code that is not a prefix of any other, the colours can be replaced by their bit codes in the image file. The most frequently occurring colour, white, will be represented with just a single bit rather than 8 bits. Black will take two bits. Red and blue will take three. After these replacements are made, the 200-bit image will be compressed to 14 x 1 + 6 x 2 + 3 x 3 + 2 x 3 = 41 bits, which is about 5 bytes compared to 25 bytes in the original image.

Of course, to decode the image the compressed file must include the code table, which takes up some space.

Lossless JPEG compression uses the Huffman algorithm in its pure form. Lossless JPEG is common in medicine as part of the **DICOM** standard, which is supported by the major medical equipment manufacturers (for use in ultrasound machines, nuclear resonance imaging machines, MRI machines, and electron microscopes).

Variations of the Lossless JPEG algorithm are also used in the RAW format, which is popular among photo enthusiasts because it saves data from a camera's image sensor without losing information.

Let us Traverse the Huffman Tree and assign codes to characters. Steps to build Huffman Tree Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

Create a leaf node for each unique character and build a min-heap of all leaf nodes (Min Heap is used as a priority queue. The value of the frequency field is used to compare two nodes in the min-heap.

Initially, the least frequent character is at the root) Extract two nodes with the minimum frequency from the min-heap.

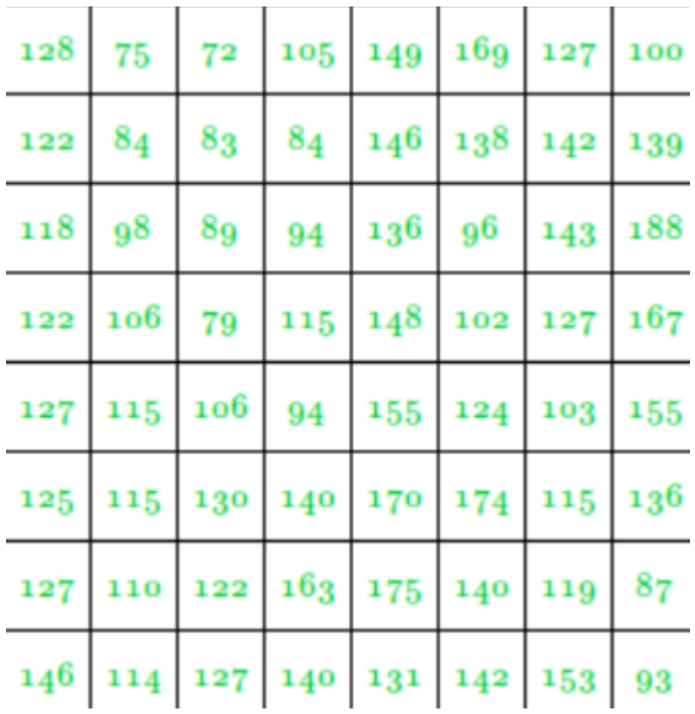
Create a new internal node with a frequency equal to the sum of the two nodes' frequencies. Make the first extracted node its left child and the other extracted node as its right child. Add this node to the min-heap.

Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

Let's take an 8 X 8 image

The pixel intensity values are:



128	75	72	105	149	169	127	100
122	84	83	84	146	138	142	139
118	98	89	94	136	96	143	188
122	106	79	115	148	102	127	167
127	115	106	94	155	124	103	155
125	115	130	140	170	174	115	136
127	110	122	163	175	140	119	87
146	114	127	140	131	142	153	93

This image contains 46 distinct pixel intensity values; hence we will have 46 unique Huffman code words.It is evident that not all pixel intensity values may be present in the image and hence will not have a non-zero probability of occurrence.From here on, the pixel intensity values in the input Image will be addressed as leaf nodes.

Now, there are 2 essential steps to build a Huffman Tree:

Build a Huffman Tree:

1. *Combine the two lowest probability leaf nodes into a new node.*
2. *Replace the two leaf nodes with the new node and sort the nodes according to the new probability values.*
3. *Continue steps (a) and (b) until we get a single node with a probability value of 1.0. We will call this node a root*

Algorithm for the problem:

Step 1: Read the image into a 2d array(image)

Step 2: Define a struct which will contain the pixel intensity values(pix), their corresponding(freq), the pointer to the left(*left) and right(*right) Child nodes and the string array for the Huffman code word(code).

Step 3: Define another Struct which will contain the pixel intensity values(pix), their corresponding probabilities(freq) and an additional field, which will be used for storing the position of new generated nodes(arrloc)

Step 4: Declaring an array of structs. Each element of the array corresponds to a node in the Huffman Tree.

Step 5:Initialize the two arrays pix_freq and huffcodes with information of the leaf nodes.

Step 6: Sorting the huffcodes array according to the probability of occurrence of the pixel intensity values.

Step 7:Building the Huffman Tree

<u>Code:</u>

```
// C Code for
// Image Compression
#include <stdio.h>
#include <stdlib.h>

// function to calculate word length
int codelen(char* code)
{
        int l = 0;
        while (*(code + l) != '\0')
                l++;
        return l;
}

// function to concatenate the words
void strconcat(char* str, char* parentcode, char add)
{
        int i = 0;
        while (*(parentcode + i) != '\0')
        {
                *(str + i) = *(parentcode + i);



                        i++;
                }
                if (add != '2')
                {
                        str[i] = add;
                        str[i + 1] = '\0';
                }
                else
                        str[i] = '\0';
        }

        // function to find fibonacci number
        int fib(int n)
        {
                if (n <= 1)
                        return n;
                return fib(n - 1) + fib(n - 2);
        }
```

```
// Driver code
int main()
{
        int i, j;
        char filename[] = "Input_Image.bmp";
        int data = 0, offset, bpp = 0, width, height;
        long bmpsize = 0, bmpdataoff = 0;
        int** image;
        int temp = 0;

        // Reading the BMP File
        FILE* image_file;

        image_file = fopen(filename, "rb");
        if (image_file == NULL)
        {
                printf("Error Opening File!!");




            exit(1);
    }
    else
    {

            // Set file position of the
            // stream to the beginning
            // Contains file signature
            // or ID "BM"
            offset = 0;

            // Set offset to 2, which
            // contains size of BMP File
            offset = 2;

            fseek(image_file, offset, SEEK_SET);

            // Getting size of BMP File
            fread(&bmpsize, 4, 1, image_file);
```

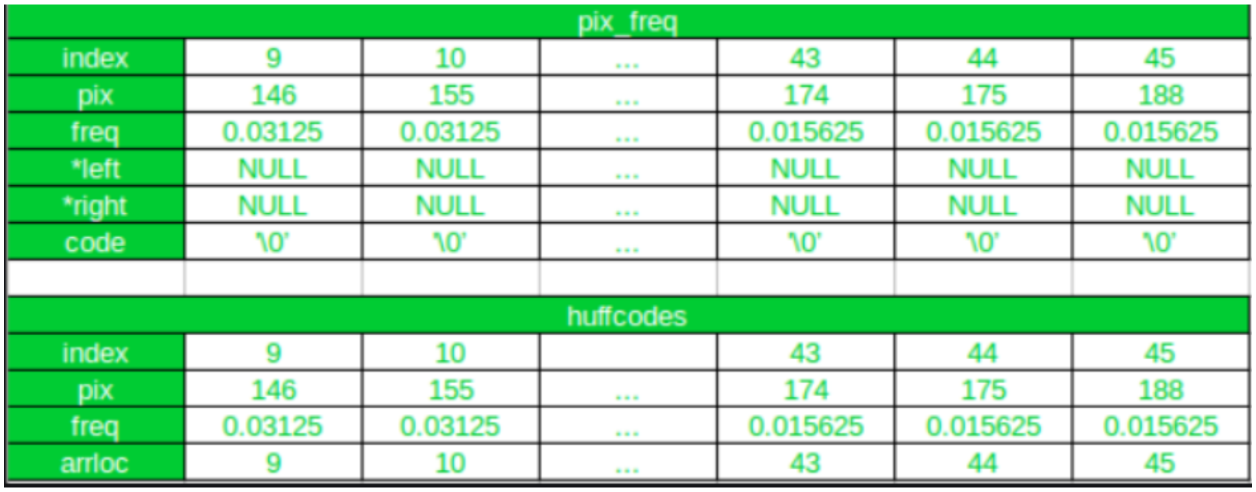
```
// Getting offset where the
// pixel array starts
// Since the information is
// at offset 10 from the start,
// as given in BMP Header
offset = 10;

fseek(image_file, offset, SEEK_SET);

// Bitmap data offset
fread(&bmpdataoff, 4, 1, image_file);

// Getting height and width of the image
// Width is stored at offset 18 and
// height at offset 22, each of 4 bytes
fseek(image_file, 18, SEEK_SET);



    fread(&width, 4, 1, image_file);

    fread(&height, 4, 1, image_file);

    // Number of bits per pixel
    fseek(image_file, 2, SEEK_CUR);

    fread(&bpp, 2, 1, image_file);

    // Setting offset to start of pixel data
    fseek(image_file, bmpdataoff, SEEK_SET);

    // Creating Image array
    image = (int**)malloc(height * sizeof(int*));

    for (i = 0; i < height; i++)
    {
        image[i] = (int*)malloc(width * sizeof(int));
    }

    // int image[height][width]
    // can also be done
    // Number of bytes in
    // the Image pixel array
    int numbytes = (bmpsize - bmpdataoff) / 3;

    // Reading the BMP File
    // into Image Array
    for (i = 0; i < height; i++)
    {
        for (j = 0; j < width; j++)
        {
            fread(&temp, 3, 1, image_file);

            // the Image is a
```

```
                        // 24-bit BMP Image
                        temp = temp & 0x0000FF;
                        image[i][j] = temp;
                }
        }
}



// Defining Structures
// huffcode
struct huffcode
{
        int pix, arrloc;
        float freq;
};

// Declaring structs
struct pixfreq* pix_freq;
struct huffcode* huffcodes;
int totalnodes = 2 * nodes - 1;
pix_freq = (struct pixfreq*)malloc(sizeof(struct pixfreq) * totalnodes);
huffcodes = (struct huffcode*)malloc(sizeof(struct huffcode) * nodes);

// Initializing
j = 0;
int totpix = height * width;
float tempprob;
for (i = 0; i < 256; i++)
{
        if (hist[i] != 0)
        {

                // pixel intensity value
                huffcodes[j].pix = i;
                pix_freq[j].pix = i;





                // location of the node
                // in the pix_freq array
                huffcodes[j].arrloc = j;

                // probability of occurrence
                tempprob = (float)hist[i] / (float)totpix;
                pix_freq[j].freq = tempprob;
                huffcodes[j].freq = tempprob;

                // Declaring the child of leaf
                // node as NULL pointer
                pix_freq[j].left = NULL;
                pix_freq[j].right = NULL;

                // initializing the code
                // word as end of line
                pix_freq[j].code[0] = '\0';
                j++;
        }
}
```

```
    // Sorting the histogram
    struct huffcode temphuff;

    // Sorting w.r.t probability
    // of occurrence
    for (i = 0; i < nodes; i++)
    {
            for (j = i + 1; j < nodes; j++)
            {
                    if (huffcodes[i].freq < huffcodes[j].freq)
                    {
                            temphuff = huffcodes[i];
                            huffcodes[i] = huffcodes[j];
                            huffcodes[j] = temphuff;


}

}}
```

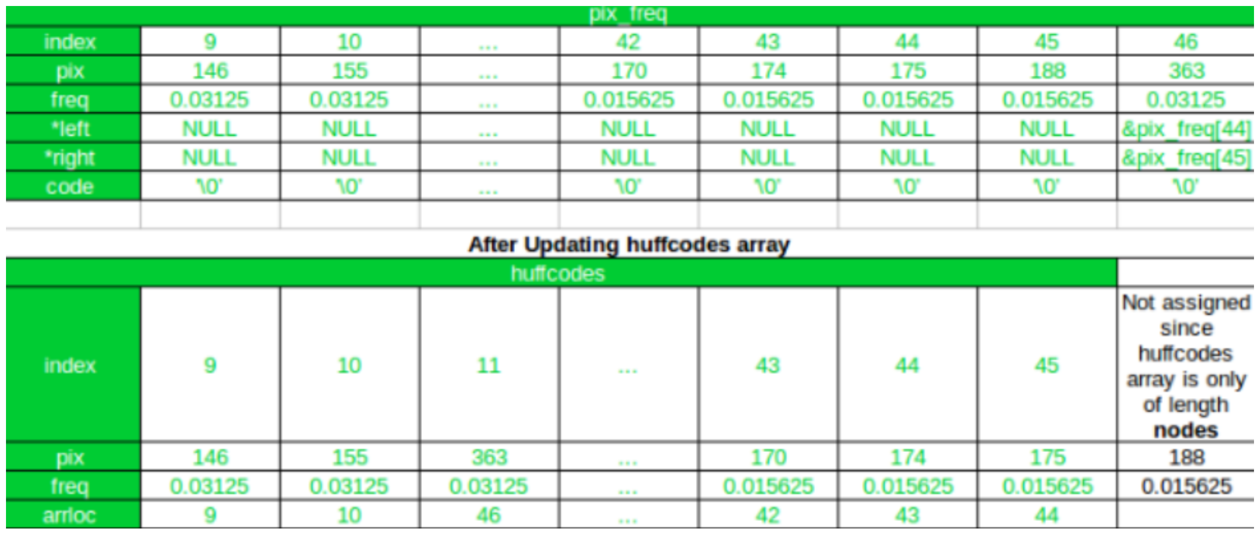
Algorithm Explanation with Example(Working):

Initially:



pix_freq						
index	9	10	...	43	44	45
pix	146	155	...	174	175	188
freq	0.03125	0.03125	...	0.015625	0.015625	0.015625
*left	NULL	NULL	...	NULL	NULL	NULL
*right	NULL	NULL	...	NULL	NULL	NULL
code	'\0'	'\0'	...	'\0'	'\0'	'\0'
huffcodes						
index	9	10		43	44	45
pix	146	155	...	174	175	188
freq	0.03125	0.03125	...	0.015625	0.015625	0.015625
arrloc	9	10	...	43	44	45

After First Iteration:



pix_freq							
index	9	10	...	42	43	44	45	46
pix	146	155	...	170	174	175	188	363
freq	0.03125	0.03125	...	0.015625	0.015625	0.015625	0.015625	0.03125
*left	NULL	NULL	...	NULL	NULL	NULL	NULL	&pix_freq[44]
*right	NULL	NULL	...	NULL	NULL	NULL	NULL	&pix_freq[45]
code	'\0'	'\0'	...	'\0'	'\0'	'\0'	'\0'	'\0'

After Updating huffcodes array

huffcodes								
index	9	10	11	...	43	44	45	Not assigned since huffcodes array is only of length **nodes**
pix	146	155	363	...	170	174	175	188
freq	0.03125	0.03125	0.03125	...	0.015625	0.015625	0.015625	0.015625
arrloc	9	10	46	...	42	43	44	

<u>Complexity Analysis:</u>

Time complexity:

O(nlog n)

The time complexity for encoding each unique character based on its frequency is O(nlog n). Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n). Thus the overall complexity is O(nlog n).

Conclusion:

Hence the image compression using the Huffman coding was completed and executed successfully.

References and Bibliography:

https://www.geeksforgeeks.org/image-compression-using-huffman-coding/

https://raw.githubusercontent.com/sudhamshu091/Huffman-Encoding-Decoding-For-Image-Compression/main/ksd.jpg

https://www.print-driver.com/stories/huffman-coding-jpeg

www.stackoverflow.com