A

PROJECT

REPORT

ON

Image Compression

Submitted in partial fulfillment of the requirement for the IV semester

of

BACHELOR OF

TECHNOLOGY

IN

DESIGN AND ANALYSIS OF ALGORITHM

(18CSC204J)

Submitted By:
Sindhu Kaleeswaran - RA2011026010082
Anindya Shankar Dasgupta - RA2011026010120
Riddhiman Bhattacharya - RA2011031010016

Under the supervision of

DR. B PANDEESWARI

(AP, CTECH)

DEPARTMENT OF COMPUTING AND TECHNOLOGY, SRM INSTITUTE OF SCIENCE AND TECHNOLOGY SESSION – 2022

SRMI INSTITUTE OF SCIENCE AND TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR -603 203

BONAFIDECERTIFICATE

Register No. RA20110260082, RA2011026010120, RA2011031010016

Certified to be bonafide record of the work done by **SINDHU KALEESWARAN, ANINDYA SHANKAR DASGUPTA, RIDDHIMAN BHATTACHARYA** *of ____________________ ________________, B.tech degree course in the practical 18CSC204J-Design and Analysis of Algorithms in SRM Institute of Science and Technology, Kattankulathur during the academic year 2021-22.*

Date: **Lab Incharge:**

Submitted for university examination held in _________________ SRM Institute of Science and Technology, Kattankulathur.

School of Computing

SRM IST, Kattankulathur – 603 203

Course Code: 18CSC204J

Course Name: Design and Analysis of Algorithm

Title of Experiment	**Image Compression through encoding using Greedy Algorithm**
Team Members	SINDHU KALEESWARAN – RA2011026010082 (Q1) ANINDYA SHANKAR DASGUPTA – RA2011026010120 (Q1) RIDDHIMAN BHATTACHARYA – RA2011031010016 (O1)
Register Number	RA2011026010082 RA2011026010120 RA2011031010016
Date of Experiment	

Staff Signature with date

Aim: To carry out image compression using pixel weights, by application of greedy algorithm.

Team Members:

S No	Register No	Name	Role
1	RA2011026010120	Anindya Shankar Dasgupta	**Rep**
2	RA2011026010082	Sindhu Kaleeswaran	**Member**
3	RA2011031010016	Riddhiman Bhattacharya	**Member**

Table of Contents

Contribution Table

Name	Reg no	Section	Contribution
Sindhu Kaleeswaran	RA2011026010082	**Q1**	Documentation, Complexity Analysis, presentation.
Anindya Shankar Dasgupta	RA2011026010120	**Q1**	Algorithm, dry-runs, methodology definitions, presentation.
Riddhiman Bhattacharya	RA2011031010016	**O1**	Coding, Compiling-Running, sample input/output, presentation.

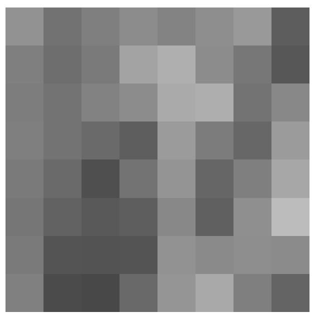
Problem Title : Image Compression

Problem Description:

Image Compression is a necessity in our daily life. As we know, every digital image is made up of pixels. Now, there may be several pixels repeated throughout the image. Thus, using a common identifier to store such pixels can drastically reduce the memory space taken up by the image. This Project will focus on working out an efficient algorithm for doing the same.

Problem Explanation:

The source symbols can be either pixel intensities of the image, or the output of an intensity mapping function. The first step of the technique is to reduce the input image to a ordered histogram, where the probability of occurrence of a certain pixel intensity value is as: **prob_pixel = numpix/totalnum,** where numpix is the number of occurrence of a pixel with a certain intensity value and totalnum is the total number of pixels in the input Image, eg:



The pixel intensities are given by a histogram:

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. It is evident that, not all pixel intensity values may be present in the image and hence will not have non-zero probability of occurrence. From here on, the pixel intensity values in the input Image will be addressed as leaf nodes. Thus, the problem will be solved based on the pixel intensities.

Design Technique used:

In this case, we will be using the **Greedy Methodology** to implement an advanced tabular encoding technique to encrypt the image pixels into a shorter format.

Greedy methodology:

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy methods are easy to implement and quite efficient in most cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

This image contains 46 distinct pixel intensity values, hence we will have 46 unique code words.
It is evident that not all pixel intensity values may be present in the image and hence will not have 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 theTree :

1. **Build a Tree** :
    1. Combine the two lowest probability leaf nodes into a new node.
    2. Replace the two leaf nodes by the new node and sort the nodes according to the new probability values.
    3. Continue the steps (a) and (b) until we get a single node with probability value 1.0. We will call this node as root
2. Backtrack from the root, assigning ‘0’ or ‘1’ to each intermediate node, till we reach the leaf nodes.

In this example, we will assign ‘0’ to the left child node and ‘1’ to the right one.

Algorithm:

Here, we list the stepwise modules of the program to carry out the image compression.

Steps:

1. Read the image into a 2D array.
   - Create a Histogram of the pixel intensity values present in the Image
   - Find the number of pixel intensity values having non-zero probability of occurrence
   - Calculating the maximum length of code words. (If p>1/Fk+3, then in any efficient prefix code for a source whose least probability is p, the longest codeword length is at most k & If p<=1/Fk+2, there exists a source whose smallest probability is p<1/Fk+2, and which has a Huffman code whose longest word has length K. If there exists such a source for which every optimal code has a longest word of length K. (Here, Fk is the kth Fibonacci number.)
2. Define a struct which will contain the pixel intensity values(pix), their corresponding probabilities(freq), the pointer to the left(*left) and right(*right) child nodes and also the string array for the code word(code).

   These structs are defined inside main(), so as to use the maximum length of code(maxcodelen) to declare the code array field of the struct pixfreq.

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

   Now, if there are N number of leaf nodes, the total number of nodes in the whole Tree will be equal to 2N-1.

   And after two nodes are combined and replaced by the new parent node, the number of nodes decreases by 1 at each iteration. Hence, it is sufficient to have a length of nodes for the array codes, which will be used as the updated and sorted nodes.

5. Initialize the two arrays **pix_freq** and **codes** with corresponding information of the leaf nodes.
6. Sorting the **codes** array according to the probability of occurrence of the pixel intensity values.

   It is necessary to sort the codes array, but not the pix_freq array, since we are already storing the location of the pixel values in the **arrloc** field of the codes array.

7. Building the Tree:

   We start by combining the two nodes with lowest probabilities of occurrence and then replacing the two nodes by the new node. This process continues until we have a root node. The first parent node formed will be stored at index nodes in the array pix_freq and the subsequent parent nodes obtained will be stored at higher values of index.

8. Backtrack from the root to the leaf nodes to assign code words

Starting from the root, we assign ‘0’ to the left child node and ‘1’ to the right child node.

Now, since we were appending the newly formed nodes to the array pix_freq, hence it is expected that the root will be the last element of the array at index totalnodes-1.

Hence, we start from the last index and iterate over the array, assigning code words to the left and right child nodes, till we reach the first parent node formed at index nodes. We don’t iterate over the leaf nodes since those nodes has NULL pointers as their left and right child.

9. Encode the Image.

10. Average number of bits required to represent each pixel is calculated:

    The function codelen calculates the length of codewords OR, the number of bits required to represent the pixel.

Why use two struct arrays?

- Initially, the struct array pix_freq, as well as the struct array codes will only contain the information of all the leaf nodes in the Tree.
- The struct array pix_freq will be used to store all the nodes of the Tree and the array codes will be used as the updated (and sorted) tree.
- Remember that, only codes will be sorted in each iteration, and not pix_freq.
- The new nodes created by combining two nodes of lowest frequency, in each iteration, will be appended to the end of the pix_freq array, and also to codes array.
- But the array codes will be sorted again according to the probability of occurrence, after the new node is added to it.
- The position of the new node in the array pix_freq will be stored in the arrloc field of the **struct** code.
- The arrloc field will be used when assigning the pointer to the left and right child of a new node.

Explanation of algorithm with example:

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 the Iteration:

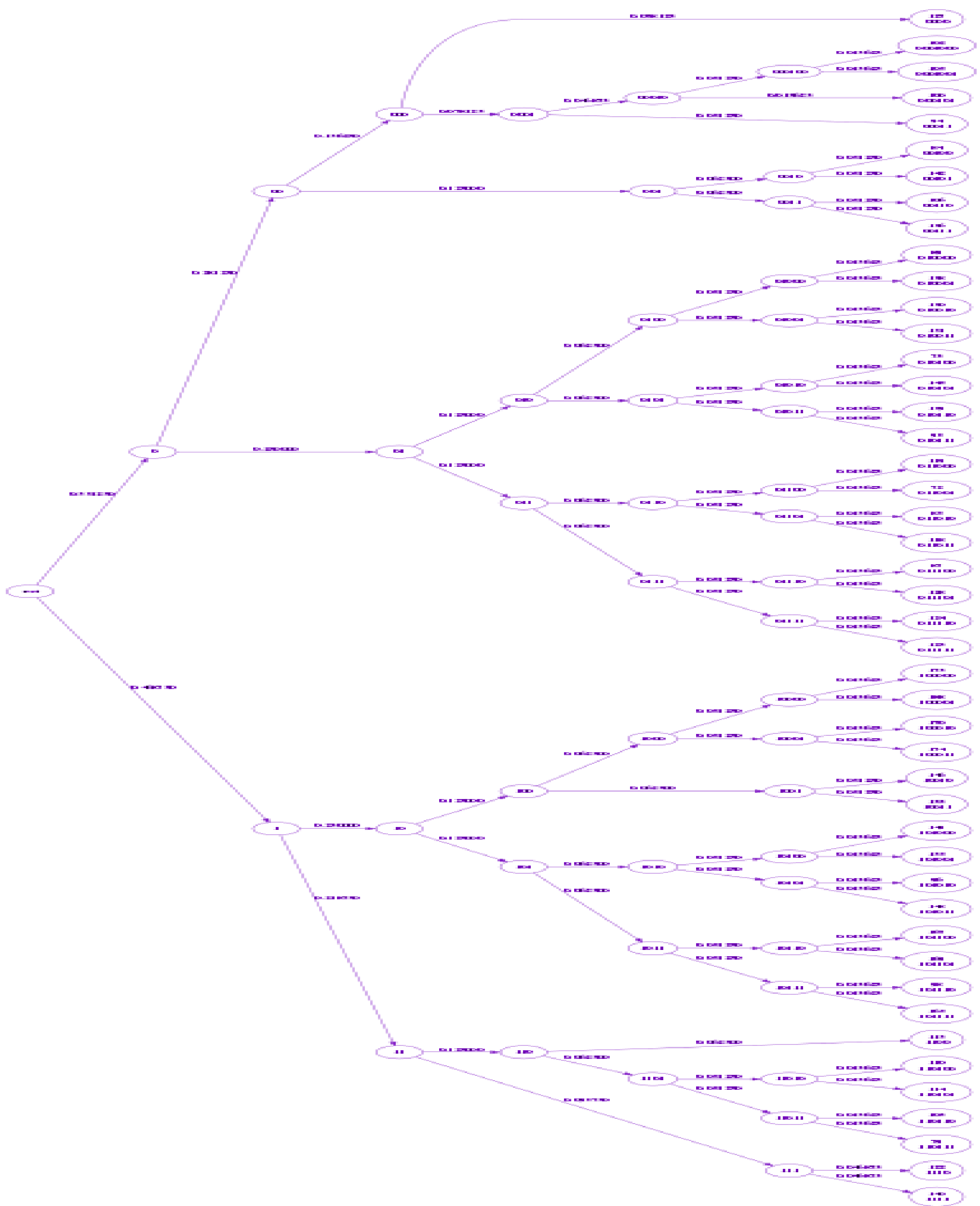
pix_freq								
index	9	10	...	42	43	44	45	46
pix	146	155	...	170	174	175	188	363
freq	0.03125	0.03125	...	0.015625	0.015625	0.015625	0.015625	0.03125
*left	NULL	NULL	...	NULL	NULL	NULL	NULL	&pix_freq[44]
*right	NULL	NULL	...	NULL	NULL	NULL	NULL	&pix_freq[45]
code	'\0'	'\0'	...	'\0'	'\0'	'\0'	'\0'	'\0'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

- After the first iteration, the new node has been appended to the pix_freq array, and its index is 46. And in the code the new node has been added at its new position after sorting, and the arrloc points to the index of the new node in the pix_freq array. Also, notice that, all array elements after the new node (at index 11) in codes array have been shifted by 1 and the array element with pixel value 188 gets excluded in the updated array.
- Now, in the next(2nd) iteration 170 and 174 will be combined, since 175 and 188 have already been combined.
- Index of the lowest two nodes in terms of the variable nodes and n is: **`left_child_index=(nodes-n-2)`** and **`right_child_index=(nodes-n-1)`**
- In the 2nd iteration, the value of n is 1 (since n starts from 0).

- For node having value 170: **`left_child_index=46-1-2=43`**
- For node having value 174: **`right_child_index=46-1-1=44`**
- Hence, even if 175 remains the last element of the updated array, it will get excluded.
- If in any subsequent iteration, the new node formed in the first iteration is the child of another new node, then the pointer to the new node obtained in the first iteration, can be accessed using the arrloc stored in huffcodes array, as is done in this line of code:

```
pix_freq[nextnode].right = &pix_freq[huffcodes[nodes - n
- 1].arrloc];
```

Tree formed:



Code(with documentation):

```c
#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);
}
```

```c
// 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);
```

```c

        // 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));
        }
```

```c

        // 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;
            }
        }
    }

    // Finding the probability
    // of occurrence
    int hist[256];
    for (i = 0; i < 256; i++)
        hist[i] = 0;
    for (i = 0; i < height; i++)
        for (j = 0; j < width; j++)
            hist[image[i][j]] += 1;

    // Finding number of
    // non-zero occurrences
    int nodes = 0;
    for (i = 0; i < 256; i++)
        if (hist[i] != 0)
```

```c
    if (hist[i] != 0)
        nodes += 1;

// Calculating minimum probability
float p = 1.0, ptemp;
for (i = 0; i < 256; i++)
{
    ptemp = (hist[i] / (float)(height * width));
    if (ptemp > 0 && ptemp <= p)
        p = ptemp;
}

// Calculating max length
// of code word
i = 0;
while ((1 / p) > fib(i))
    i++;
int maxcodelen = i - 3;

// Defining Structures pixfreq
struct pixfreq
{
    int pix, larrloc, rarrloc;
    float freq;
    struct pixfreq *left, *right;
    char code[maxcodelen];
};

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

// Declaring structs
```

```c
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++;
```

```c
            // 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;
            }
        }
    }

    // Building Tree
    float sumprob;
    int sumpix;
    int n = 0, k = 0;
    int nextnode = nodes;

    // Since total number of
```

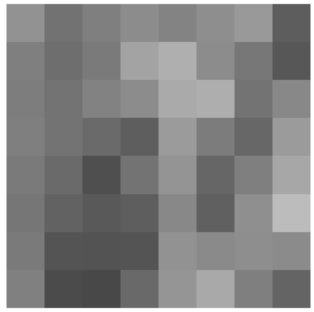
```c
    // Building Tree
    float sumprob;
    int sumpix;
    int n = 0, k = 0;
    int nextnode = nodes;

    // Since total number of
    // nodes in Tree
    // is 2*nodes-1
    while (n < nodes - 1)
    {

        // Adding the lowest two probabilities
        sumprob = huffcodes[nodes - n - 1].freq + huffcodes[nodes - n - 2].freq;
        sumpix = huffcodes[nodes - n - 1].pix + huffcodes[nodes - n - 2].pix;

        // Appending to the pix_freq Array
        pix_freq[nextnode].pix = sumpix;
        pix_freq[nextnode].freq = sumprob;
        pix_freq[nextnode].left = &pix_freq[huffcodes[nodes - n - 2].arrloc];
        pix_freq[nextnode].right = &pix_freq[huffcodes[nodes - n - 1].arrloc];
        pix_freq[nextnode].code[0] = '\0';
        i = 0;

        // Sorting and Updating the
        // codes array simultaneously
        // New position of the combined node
        while (sumprob <= huffcodes[i].freq)
              i++;

        // Inserting the new node
        // in the codes array
        for (k = nodes; k >= 0; k--)
        {
            if (k == i)
            {
                huffcodes[k].pix = sumpix;
```

```c
            }
            else if (k > i)
            {
                // Shifting the nodes below
                // the new node by 1
                // For inserting the new node
                // at the updated position k
                huffcodes[k] = huffcodes[k - 1];

            }
        }
        n += 1;
        nextnode += 1;
    }

    // Assigning Code through
    // backtracking
    char left = '0';
    char right = '1';
    int index;
    for (i = totalnodes - 1; i >= nodes; i--)
    {
        if (pix_freq[i].left != NULL)
            strconcat(pix_freq[i].left->code, pix_freq[i].code, left);
        if (pix_freq[i].right != NULL)
            strconcat(pix_freq[i].right->code, pix_freq[i].code, right);
    }

    // Encode the Image
    int pix_val;
    int l;

    // Writing the encoded
    // Image into a text file
    FILE* imagehuff = fopen("encoded_image.txt", "wb");
    for (i = 0; i < height; i++)
        for (j = 0; j < width; j++)
        {
```

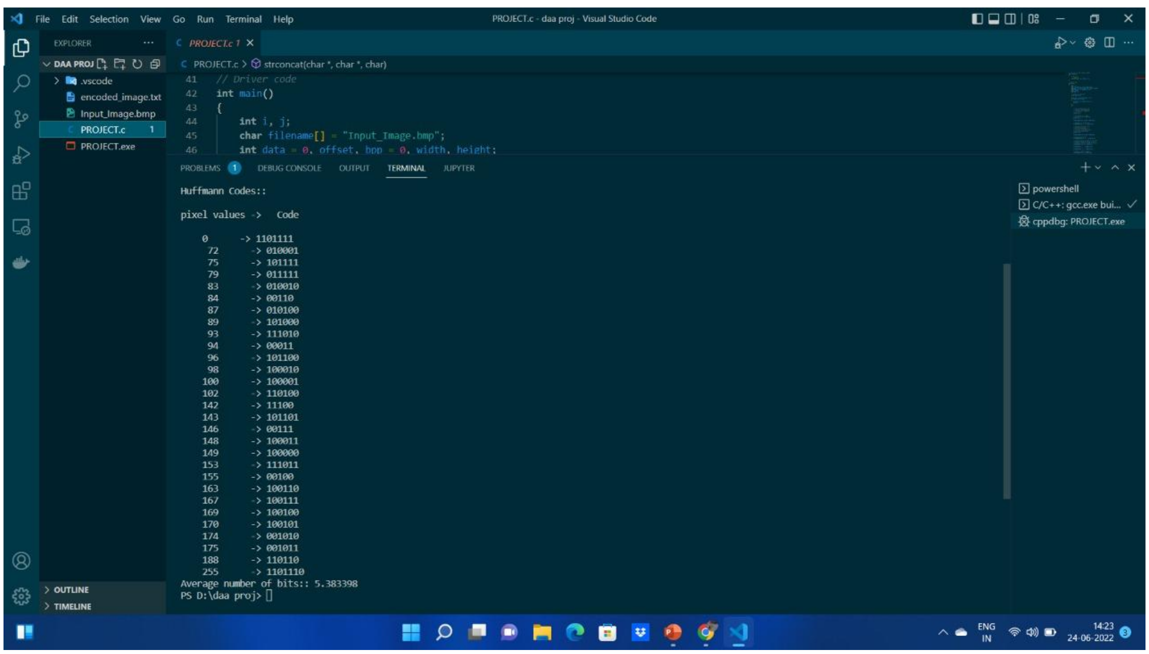
```c
            strconcat(pix_freq[i].right->code, pix_freq[i].code, right);
    }

    // Encode the Image
    int pix_val;
    int l;

    // Writing the encoded
    // Image into a text file
    FILE* imagehuff = fopen("encoded_image.txt", "wb");
    for (i = 0; i < height; i++)
        for (j = 0; j < width; j++)
        {
            pix_val = image[i][j];
            for (l = 0; l < nodes; l++)
                if (pix_val == pix_freq[l].pix)
                    fprintf(imagehuff, "%s", pix_freq[l].code);
        }

    // Printing Codes
    printf("Huffmann Codes::\n\n");
    printf("pixel values ->   Code\n\n");
    for (i = 0; i < nodes; i++)
    {
        if (snprintf(NULL, 0, "%d", pix_freq[i].pix) == 2)
            printf("     %d      -> %s\n", pix_freq[i].pix, pix_freq[i].code);
        else
            printf("    %d      -> %s\n", pix_freq[i].pix, pix_freq[i].code);
    }

    // Calculating Average Bit Length
    float avgbitnum = 0;
    for (i = 0; i < nodes; i++)
        avgbitnum += pix_freq[i].freq * codelen(pix_freq[i].code);
    printf("Average number of bits:: %f", avgbitnum);
}
```

Sample input/output:

1. **Input image:**



2. **Output:**



3. **Encoded image codes:**

0111010101000110011101101010001011010000000101111

00010001101000100100100100010010101011001101110111001

00000001100111101010010101100001111000110110111110010

101100010000000101100000001100001100001110011011110000

100110011011111110001001011111100010100011110000111000

011010011101011111100000111101100001110010010110101000

0111101001100101101001010111

Complexity Analysis:

1. **Time complexity = T(n)**

   = Time to read image + time to convert into histogram + time for encoding

   = O(n) + O(n) + O(nlogn)

   = O(nlogn)

   Total asymptotic max upper bound time complexity = O(nlogn),

   where, n is the no. of pixel weights the image has been divided into.

2. **Auxiliary space :**

   Space required for the program =

   approximate no of variables used + loops + other standalone statements

   = 25 + 8*n + 145

   = 170 + 8n

   Total asymptotic max upper bound space complexity = O(n).

Result:
Compression Ratio gives a measure of how successful this algorithm actually is in reducing the size of the image.
No. of bits required per pixel in original image = p1
2^p1 = 64 p1 = 6
No. of bits required per pixel in compressed image (shown in output) = 5.38
Compression Ratio = (no. of bits required for original image - no. of bits required for compressed) / no. of bits of original image
= (6 - 5.38)/6 = 0.1033
So, compressed by 1.033%.

Conclusion:

Thus, we have successfully constructed an algorithm, which can be used to formulate a driver code in order to carry out image compression on any .bmp images. The greedy algorithm formulates an encoded image, based on the pixel weight histogram of the image. Thus, the image gets successfully encoded, and can be decoded based on the new pixel codes and frequencies to form the compressed image.

Hence, we can conclude that an algorithm has been successfully developed which is feasible, efficient and can be solved in finite time and space to produce the desired output, i.e. the compressed image.

References:

www.geeksforgeeks.com