**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: /0 /2023**

**Experiment No. 14**

# Title: Huffman Cod

**Theory/Description of the Problem Statement:**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are Prefix Codes, means 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.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

**Algorithm :**

* create a node structure for a min heap
* - character data
* - unsigned frequency
* - left and right MinHeapNode pointers
* create a comparison function for MinHeapNodes
* - return true if left node's frequency is greater than right node's frequency
* create a function to print Huffman codes
* - if the node is null, return
* - if the node's data is not '$', print the character and its code
* - recursively call the function on the left child, adding "0" to the code string
* - recursively call the function on the right child, adding "1" to the code string
* create a function to build a Huffman tree and print the codes
* - create an empty priority queue of MinHeapNode pointers
* - insert each character-frequency pair into the priority queue as a new MinHeapNode
* - while the size of the priority queue is not 1
* - extract the two minimum frequency nodes from the priority queue
* - create a new internal node with '$' as its character data and a frequency equal to the sum of the two extracted nodes' frequencies
* - set the left and right children of the new node to the two extracted nodes
* - add the new node to the priority queue
* - call the printCodes function on the root of the Huffman tree

**Pseudo Code :**

* Algorithm Huffman (c)
* {
* n= |c|
* Q = c
* for i<-1 to n-1
* do
* {
* temp <- get node ()
* left (temp] Get\_min (Q) right [temp] Get Min (Q)
* a = left [templ b = right [temp]
* F [temp]<- f[a] + [b]
* insert (Q, temp)
* }
* return Get\_min (0)
* }

**Analysis of the Algorithm**

**Time Complexity:**

In terms of time complexity, building the Huffman tree takes O(n log n) time because it requires sorting the character-frequency pairs in a priority queue. Printing the Huffman codes takes O(n) time because it requires traversing the entire tree once. Therefore, the overall time complexity is O(n log n).

**Space Complexity:**

In terms of space complexity, the program uses O(n) space to store the character-frequency pairs and the Huffman tree. Additionally, it uses O(log n) space for the call stack during the recursive function calls to print the codes. Therefore, the overall space complexity is O(n + log n).

**Experiment and result:**

Code:

// C++(STL) program for Huffman Coding with STL

#include <bits/stdc++.h>

using namespace std;

// A Huffman tree node

struct MinHeapNode {

    // One of the input characters

    char data;

    // Frequency of the character

    unsigned freq;

    // Left and right child

    MinHeapNode \*left, \*right;

    MinHeapNode(char data, unsigned freq)

    {

        left = right = NULL;

        this->data = data;

        this->freq = freq;

    }

};

// For comparison of

// two heap nodes (needed in min heap)

struct compare {

    bool operator()(MinHeapNode\* l, MinHeapNode\* r)

    {

        return (l->freq > r->freq);

    }

};

// Prints huffman codes from

// the root of Huffman Tree.

void printCodes(struct MinHeapNode\* root, string str)

{

    if (!root)

        return;

    if (root->data != '$')

        cout << root->data << ": " << str << "\n";

    printCodes(root->left, str + "0");

    printCodes(root->right, str + "1");

}

// The main function that builds a Huffman Tree and

// print codes by traversing the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

    struct MinHeapNode \*left, \*right, \*top;

    // Create a min heap & inserts all characters of data[]

    priority\_queue<MinHeapNode\*, vector<MinHeapNode\*>,

                compare>

        minHeap;

    for (int i = 0; i < size; ++i)

        minHeap.push(new MinHeapNode(data[i], freq[i]));

    // Iterate while size of heap doesn't become 1

    while (minHeap.size() != 1) {

        // Extract the two minimum

        // freq items from min heap

        left = minHeap.top();

        minHeap.pop();

        right = minHeap.top();

        minHeap.pop();

        // Create a new internal node with

        // frequency equal to the sum of the

        // two nodes frequencies. Make the

        // two extracted node as left and right children

        // of this new node. Add this node

        // to the min heap '$' is a special value

        // for internal nodes, not used

        top = new MinHeapNode('$',

                            left->freq + right->freq);

        top->left = left;

        top->right = right;

        minHeap.push(top);

    }

    // Print Huffman codes using

    // the Huffman tree built above

    printCodes(minHeap.top(), "");

}

// Driver Code

int main()

{

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

    int freq[] = { 5, 9, 12, 13, 16, 45 };

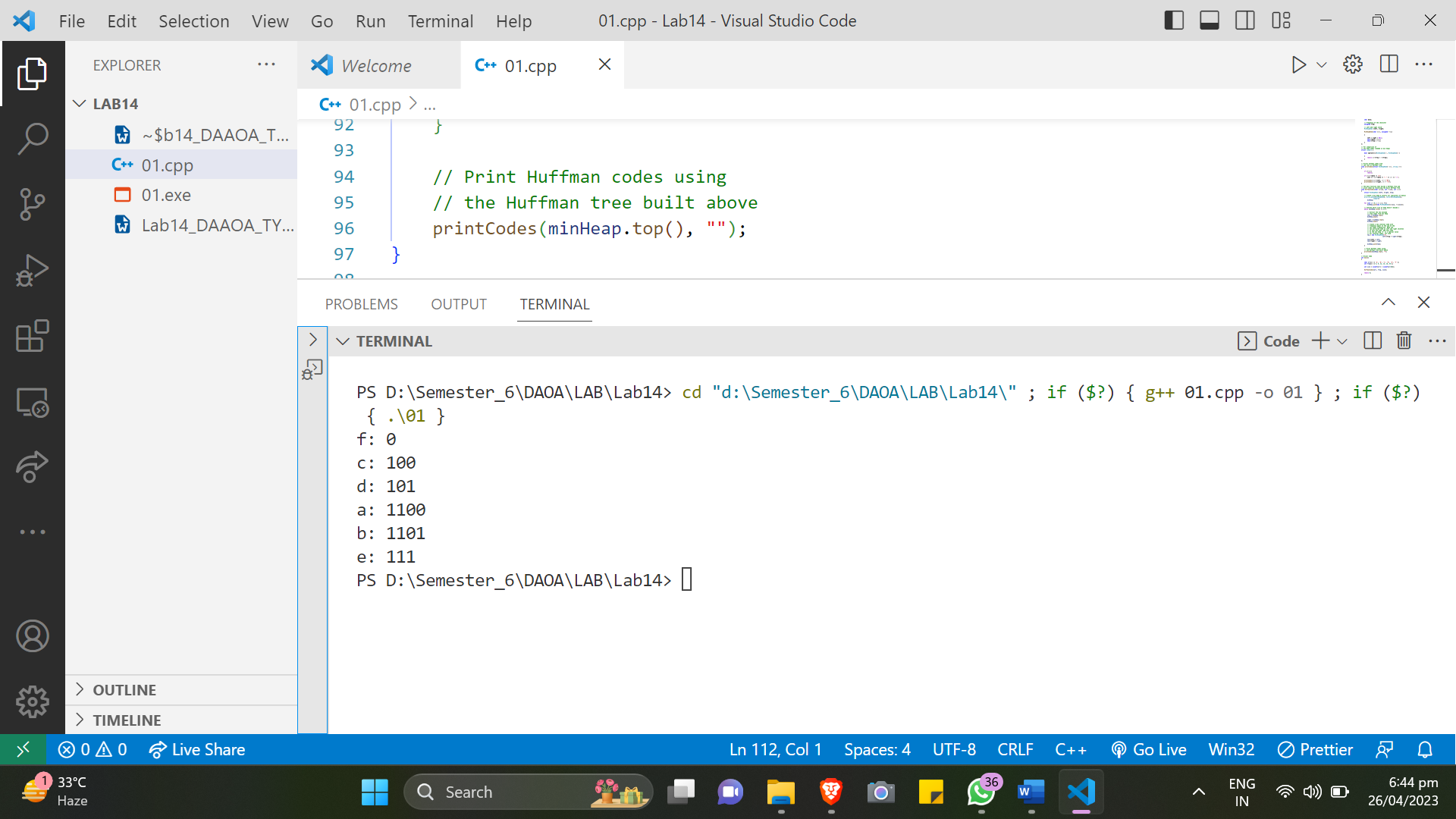
    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;

}

Output:



**Conclusions:**

Huffman coding technique to compress data by assigning variable length codes to input characters based on their frequencies. The time complexity of the program is O(nlogn), where n is the number of characters in the input, due to the use of a priority queue to build the Huffman tree. The space complexity is O(n), as the program creates a MinHeapNode for each input character.