Huffman Coding

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 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 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".

See this for applications of Huffman Coding.
There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. 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.

1. 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 frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.

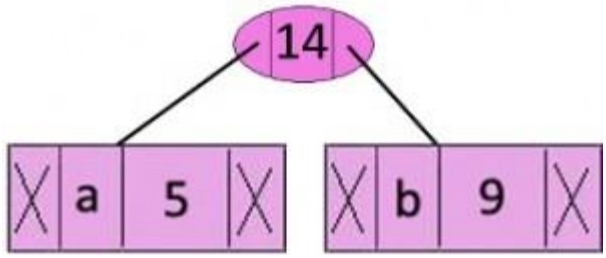
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. 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:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

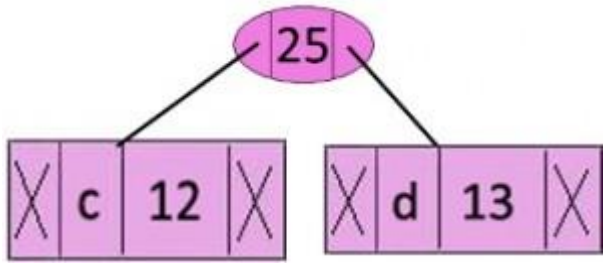
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

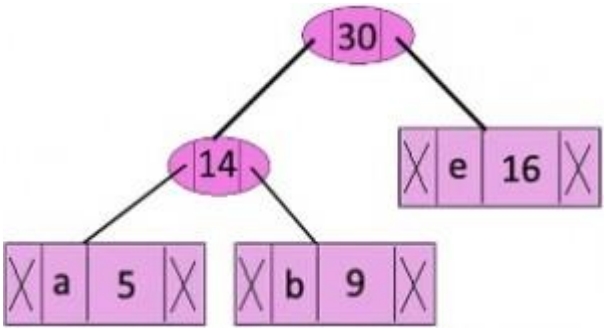
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

```
character            Frequency
Internal Node           14
       e                16
Internal Node           25
       f                45
```

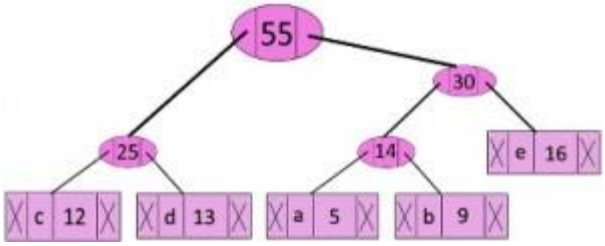
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

```
character          Frequency
Internal Node         25
Internal Node         30
      f               45
```

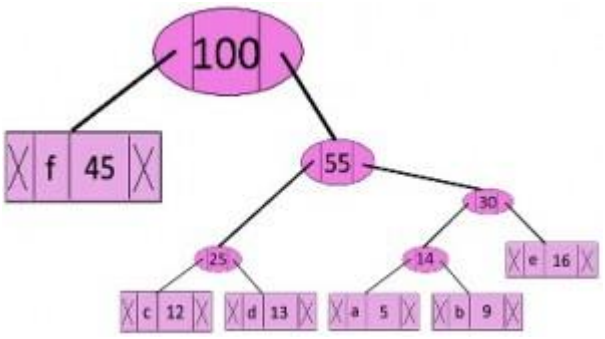
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



Now min heap contains 2 nodes.

```
character      Frequency
       f          45
Internal Node     55
```

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



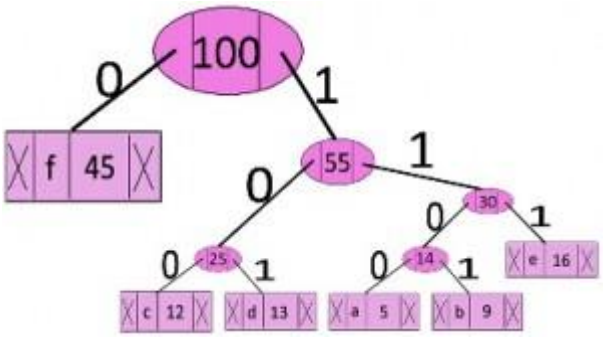
Now min heap contains only one node.

```
character       Frequency
Internal Node     100
```

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character    code-word
    f           0
    c           100
    d           101
    a           1100
    b           1101
    e           111
```

Huffman Coding using Priority Queue

Prerequisite: Greedy Algorithms | Set 3 (Huffman Coding), priority_queue::push() and priority_queue::pop() in C++ STL
Given a char array **ch[]** and frequency of each character as **freq[]**. The task is to find Huffman Codes for every character in **ch[]** using Priority Queue.

Example
Input: *ch[] = { 'a', 'b', 'c', 'd', 'e', 'f' }, freq[] = { 5, 9, 12, 13, 16, 45 }*
Output:
f 0
c 100
d 101
a 1100
b 1101
e 111

Approach:
1. Push all the characters in **ch[]** mapped to corresponding frequency **freq[]** in priority queue.
2. To create Huffman Tree, pop two nodes from priority queue.

3. Assign two popped node from priority queue as left and right child of new node.
4. Push the new node formed in priority queue.
5. Repeat all above steps until size of priority queue becomes 1.
6. Traverse the Huffman Tree (whose root is the only node left in the priority queue) to store the Huffman Code
7. Print all the stored Huffman Code for every character in **ch[]**.

Below is the implementation of the above approach:

```
// C++ Program for Huffman Coding
// using Priority Queue
#include <iostream>
#include <queue>
using namespace std;

// Maximum Height of Huffman Tree.
#define MAX_SIZE 100

class HuffmanTreeNode {
public:
    // Stores character
    char data;
    int freq;
    HuffmanTreeNode* left;
    HuffmanTreeNode* right;

    // Initializing the current node
    HuffmanTreeNode(char character, int frequency)
    {
        data = character;
        freq = frequency;
        left = right = NULL;
    }
};


 // Custom comparator class
class Compare {
public:
    bool operator()(HuffmanTreeNode* a, HuffmanTreeNode* b)
    {
        // Defining priority on the basis of frequency
        return a->freq > b->freq;
    }
};
```

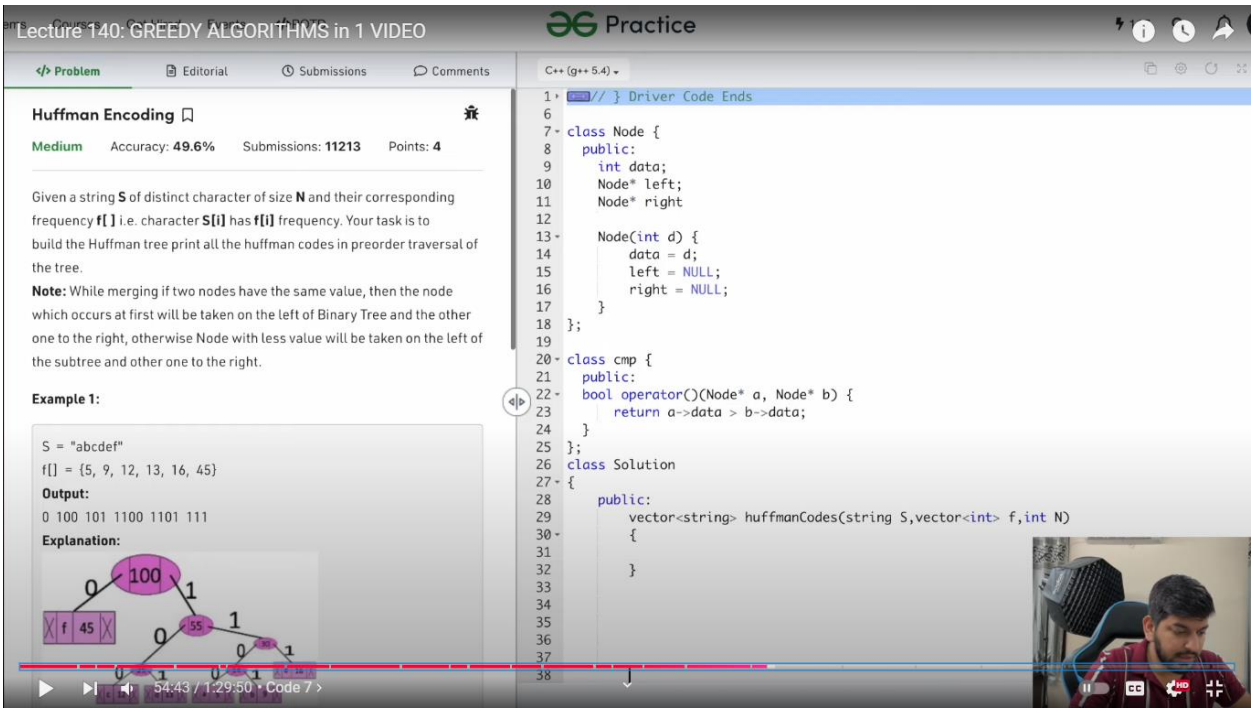
```
// Function to generate Huffma Encoding Tree
HuffmanTreeNode* generateTree(priority_queue<HuffmanTreeNode*,
                              vector<HuffmanTreeNode*>,Compare> pq)
{
    // We keep on looping till only one node remains in the Priority Queue
    while (pq.size() != 1) {

        // Node which has least frequency and Remove node from Priority Queue

        HuffmanTreeNode* left = pq.top();
        pq.pop();

        // Node which has least frequency and Remove node from Priority Queue

        HuffmanTreeNode* right = pq.top();
        pq.pop();

        // A new node is formed with frequency left->freq + right->freq
        // We take data as '$' because we are only concerned with the frequency

        HuffmanTreeNode* node = new HuffmanTreeNode('$', left->freq+ right->freq);
        node->left = left;
        node->right = right;

        // Push back node created to the Priority Queue
        pq.push(node);
    }
    return pq.top();
}

// Function to print the huffman code for each character.
// It uses arr to store the codes

void printCodes(HuffmanTreeNode* root, int arr[], int top)
{
    // Assign 0 to the left node and recur
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to the right node and recur
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then we print root->data

    // We also print the code for this character from arr
    if (!root->left && !root->right) {
        cout << root->data << " ";
```

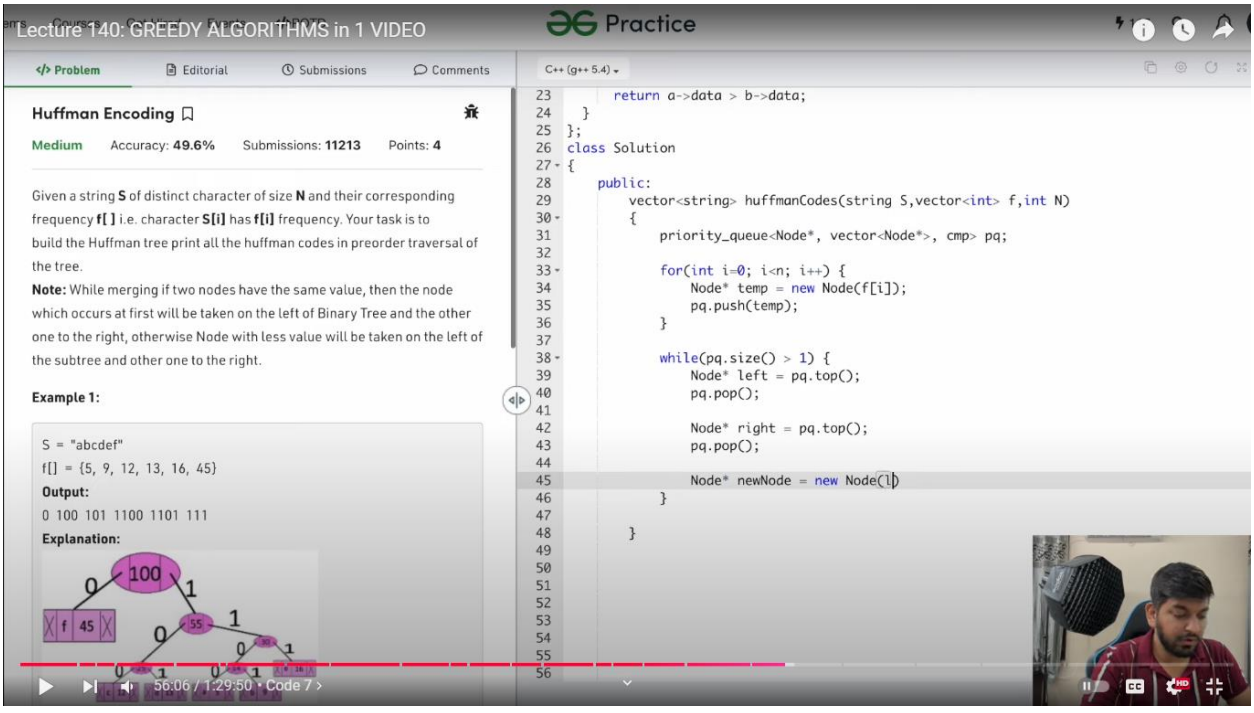
```
        for (int i = 0; i < top; i++) {
            cout << arr[i];
        }
        cout << endl;
    }
}

void HuffmanCodes(char data[],int freq[], int size)
{
    // Declaring priority queue using custom comparator
    priority_queue<HuffmanTreeNode*,vector<HuffmanTreeNode*>,Compare> pq;

    // Populating the priority queue
    for (int i = 0; i < size; i++) {
        HuffmanTreeNode* newNode = new HuffmanTreeNode(data[i], freq[i]);
        pq.push(newNode);
    }

    // Generate Huffman Encoding Tree and get the root node
    HuffmanTreeNode* root = generateTree(pq);

    // Print Huffman Codes
    int arr[MAX_SIZE], top = 0;
    printCodes(root, arr, top);
}

// Driver Code
int main()
{
    char data[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };
    int size = sizeof(data) / sizeof(data[0]);

    HuffmanCodes(data, freq, size);
    return 0;
}
```

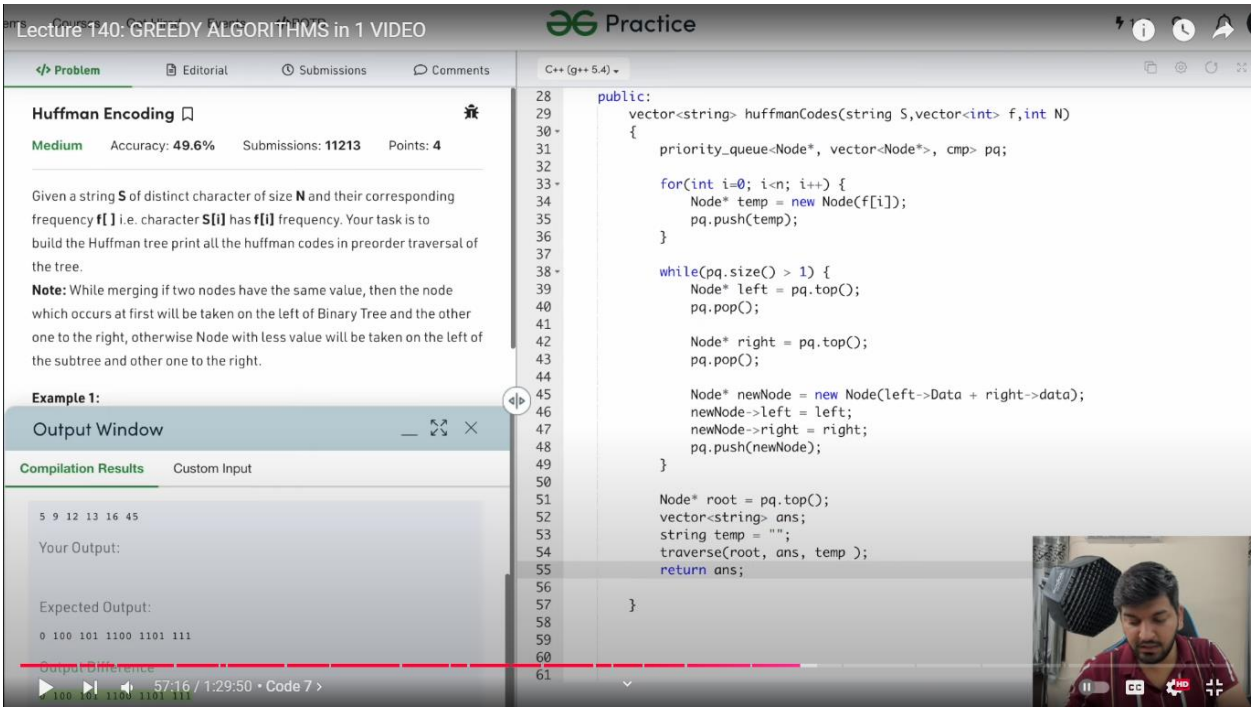
Output:

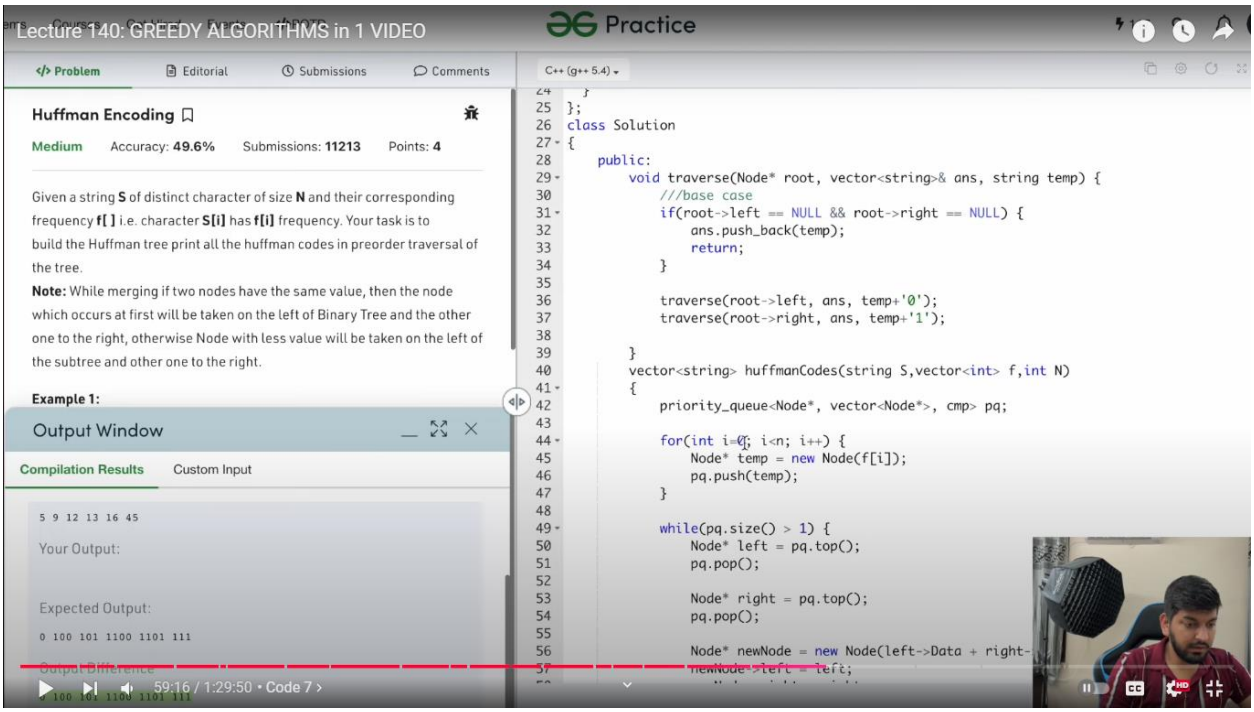
```
f 0
c 100
d 101
a 1100
b 1101
e 111
```

Time Complexity: O(n*logn) where n is the number of unique characters
Auxiliary Space: O(n)






Lecture 140: GREEDY ALGORITHMS in 1 VIDEO
Practice
Problem
Editorial
Submissions
Comments
Huffman Encoding
Medium
Accuracy: 49.6%
Submissions: 11213
Points: 4
Given a string S of distinct character of size N and their corresponding frequency f[ ] i.e. character S[i] has f[i] frequency. Your task is to build the Huffman tree print all the huffman codes in preorder traversal of the tree.
Note: While merging if two nodes have the same value, then the node which occurs at first will be taken on the left of Binary Tree and the other one to the right, otherwise Node with less value will be taken on the left of the subtree and other one to the right.
Example 1:
Output Window
Compilation Results
Custom Input
5 9 12 13 16 45
Your Output:
Expected Output:
0 100 101 1100 1101 111
Output Difference
57:16 / 1:29:50 • Code 7 ›
C++ (g++ 5.4)
public:
vector<string> huffmanCodes(string S,vector<int> f,int N)
priority_queue<Node*, vector<Node*>, cmp> pq;
for(int i=0; i<n; i++) {
Node* temp = new Node(f[i]);
pq.push(temp);
while(pq.size() > 1) {
Node* left = pq.top();
pq.pop();
Node* right = pq.top();
pq.pop();
Node* newNode = new Node(left->Data + right->data);
newNode->left = left;
newNode->right = right;
pq.push(newNode);
Node* root = pq.top();
vector<string> ans;
string temp = "";
traverse(root, ans, temp);
return ans;


Lecture 140: GREEDY ALGORITHMS in 1 VIDEO
Practice
Problem
Editorial
Submissions
Comments
Huffman Encoding
Medium
Accuracy: 49.6%
Submissions: 11213
Points: 4
Given a string S of distinct character of size N and their corresponding frequency f[ ] i.e. character S[i] has f[i] frequency. Your task is to build the Huffman tree print all the huffman codes in preorder traversal of the tree.
Note: While merging if two nodes have the same value, then the node which occurs at first will be taken on the left of Binary Tree and the other one to the right, otherwise Node with less value will be taken on the left of the subtree and other one to the right.
Example 1:
Output Window
Compilation Results
Custom Input
5 9 12 13 16 45
Your Output:
Expected Output:
0 100 101 1100 1101 111
Output Difference
59:16 / 1:29:50 • Code 7 ›
C++ (g++ 5.4)
};
class Solution
{
public:
void traverse(Node* root, vector<string>& ans, string temp) {
///base case
if(root->left == NULL && root->right == NULL) {
ans.push_back(temp);
return;
traverse(root->left, ans, temp+'0');
traverse(root->right, ans, temp+'1');
vector<string> huffmanCodes(string S,vector<int> f,int N)
priority_queue<Node*, vector<Node*>, cmp> pq;
for(int i=0; i<n; i++) {
Node* temp = new Node(f[i]);
pq.push(temp);
while(pq.size() > 1) {
Node* left = pq.top();
pq.pop();
Node* right = pq.top();
pq.pop();
Node* newNode = new Node(left->Data + right-