Lab Manual:

Course: Computer Algorithm Lab
Department of CSE, AMUST

Task 1: Huffman Coding Algorithm

1. Objective

To understand and implement Huffman Coding Algorithm for lossless data compression using C++.

2. Theory

Huffman Coding is a greedy algorithm used for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. The algorithm builds a binary tree based on the frequencies of characters.

Key Properties:

- Builds an optimal prefix code.

- Uses a priority queue (min-heap) to build the tree.

- Efficient in compressing data with a known frequency distribution.

3. Algorithm Steps

1. Create a leaf node for each character and add it to a priority queue.
2. While there is more than one node in the queue:
   a. Remove the two nodes with the lowest frequency.
   b. Create a new internal node with these two nodes as children and with frequency equal to the sum of the two.
   c. Add the new node back to the priority queue.
3. The remaining node is the root of the Huffman Tree.
4. Traverse the tree to assign binary codes to characters.

4. Sample Code in C++

```
#include <iostream>
#include <queue>
#include <unordered_map>
```

```
#include <vector>

using namespace std;

struct Node {
   char ch;
   int freq;
   Node *left, *right;

   Node(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

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

void printCodes(Node* root, string code) {
   if (!root)
      return;

   if (!root->left && !root->right) {
      cout << root->ch << ": " << code << '\n';
   }

   printCodes(root->left, code + "0");
   printCodes(root->right, code + "1");
}

void huffmanCoding(unordered_map<char, int>& freqMap) {
   priority_queue<Node*, vector<Node*>, Compare> pq;

   for (auto pair : freqMap) {
      pq.push(new Node(pair.first, pair.second));
   }

   while (pq.size() > 1) {
      Node* left = pq.top(); pq.pop();
      Node* right = pq.top(); pq.pop();

      Node* combined = new Node('\0', left->freq + right->freq);
      combined->left = left;
```

```
        combined->right = right;

        pq.push(combined);
    }

    Node* root = pq.top();
    printCodes(root, "");
}

int main() {
    unordered_map<char, int> freqMap = {
        {'a', 5}, {'b', 9}, {'c', 12},
        {'d', 13}, {'e', 16}, {'f', 45}
    };

    huffmanCoding(freqMap);
    return 0;
}
```

5. Sample Input/Output

Input: Character frequencies {a:5, b:9, c:12, d:13, e:16, f:45}

Output:
a: 1100
b: 1101
c: 100
d: 101
e: 111
f: 0 (may vary depending on tree structure)

6. Observation Table

Record the Huffman codes for different sets of characters and frequencies.

7. Viva Questions

- What is Huffman Coding used for?
- What is the advantage of using variable-length codes?
- Why is a priority queue used in Huffman Coding?
- What is a prefix code?
- Can Huffman Coding be used for real-time compression? Why or why not?

Task 2: Find max and min value of an array using divide and conquer approach

Do it yourself

Task 3: Counting Sort

1. Objective

To understand and implement Counting Sort algorithm using C++ for sorting integers within a known range.

2. Theory

Counting Sort is a non-comparison-based sorting algorithm that works best when the range of input data is not significantly larger than the number of elements to be sorted. It counts the number of occurrences of each distinct element and uses this information to place the elements in the correct position in the output array.

Key Features:

- Time complexity: O(n + k), where n is the number of elements and k is the range of input.

- Stable sort: Maintains the relative order of equal elements.

- Suitable for integers or objects with integer keys.

3. Algorithm Steps

1. Find the maximum value in the input array.
2. Initialize a count array of size (max + 1) with all zeros.
3. Count the occurrences of each element in the input array and store it in the count array.
4. Modify the count array such that each element at index i stores the sum of previous counts.
5. Build the output array by placing the elements at their correct positions using the count array.
6. Copy the sorted output array back into the original array.

4. Sample Code in C++

```
#include <iostream>
```

```
#include <vector>
using namespace std;

void countingSort(vector<int>& arr) {
   int maxVal = *max_element(arr.begin(), arr.end());
   vector<int> count(maxVal + 1, 0);
   vector<int> output(arr.size());

   for (int i : arr)
      count[i]++;

   for (int i = 1; i <= maxVal; i++)
      count[i] += count[i - 1];

   for (int i = arr.size() - 1; i >= 0; i--) {
      output[count[arr[i]] - 1] = arr[i];
      count[arr[i]]--;
   }

   arr = output;
}

int main() {
   vector<int> data = {4, 2, 2, 8, 3, 3, 1};
   countingSort(data);
   for (int num : data)
      cout << num << " ";
   return 0;
}
```

5. Sample Input/Output

Input: 4 2 2 8 3 3 1

Output: 1 2 2 3 3 4 8

6. Observation Table

Use this section to record the steps and verify correctness of the algorithm for different inputs.

7. Viva Questions

- What is the time and space complexity of Counting Sort?
- When is Counting Sort preferred over comparison-based sorting algorithms?
- Is Counting Sort stable? Justify your answer.
- Can Counting Sort be used for sorting negative numbers?
- How does Counting Sort differ from Radix Sort?