Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 09

Experiment Date: 28.05.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 2nd Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 04.06.25

Task 01: A C++ program that will implement the Huffman Coding Algorithm.

Theory:

Huffman Coding is a popular algorithm for lossless data compression. It efficiently reduces the size of data by assigning shorter binary codes to more frequent characters and longer codes to less frequent ones. This variable-length encoding ensures minimal space usage while preserving all original information.

The algorithm relies on constructing a binary tree, known as a Huffman Tree. The step-by-step outline of how it works:

1. Create a leaf node for each character and insert it into a priority queue based on frequency.
2. While there is more than one node in the queue:
    1. Remove the two nodes with the smallest frequencies.
    2. Create a new internal node by combining these two nodes, with its frequency being their sum.
    3. Insert the new node back into the priority queue.
3. The final remaining node in the queue becomes the root of the Huffman Tree.
4. Traverse the tree to assign binary codes to each character—assigning '0' for a left edge and '1' for a right edge.

Huffman codes are prefix-free, meaning no code is a prefix of another. This property ensures that the encoded data can be uniquely and correctly decoded without ambiguity.

Code:

```
#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;
}
```

Output:

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

Process returned 0 (0x0)   execution time : 0.121 s
Press any key to continue.
```

Conclusion:

The implementation of Huffman Coding successfully demonstrates how frequency-based compression optimizes data storage. By constructing a priority queue and binary tree, the algorithm efficiently generates prefix-free binary codes. This not only validates its theoretical basis but also highlights the practical benefits of reducing redundancy in text data. Overall, Huffman Coding stands out as a fundamental concept in the field of data compression.

Task 02: A C++ program that will find max and min value of an array using divide and conquer approach.

Theory:

The Divide and Conquer approach is a strategic algorithmic technique that breaks a problem into smaller sub-problems, solves each sub-problem recursively, and then combines the results. When applied to finding the minimum and maximum values in an array, this method enhances efficiency compared to brute-force linear scanning.

Instead of comparing every element individually, the array is processed in pairs—comparing two elements at a time. For each pair, the smaller one is compared with the current minimum, and the larger one is compared with the current maximum. This reduces the total number of comparisons needed, making the algorithm significantly faster, especially for large datasets. By minimizing redundant comparisons, the divide and conquer approach strikes a balance between time efficiency and clarity of logic.

Code:

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

struct Pair
{
   int min;
   int max;
};

struct Pair getMinMax(int arr[], int n)
{
   struct Pair minmax;
   int i;

   if (n % 2 == 0)
   {
      if (arr[0] > arr[1])
      {
         minmax.max = arr[0];
         minmax.min = arr[1];
      }
      else
      {
         minmax.min = arr[0];
         minmax.max = arr[1];
```

```
        }

        i = 2;
    }
    else
    {
        minmax.min = arr[0];
        minmax.max = arr[0];

        i = 1;
    }

    while (i < n - 1)
    {
        if (arr[i] > arr[i + 1])
        {
            if(arr[i] > minmax.max)
            {
                minmax.max = arr[i];
            }

            if(arr[i + 1] < minmax.min)
            {
                minmax.min = arr[i + 1];
            }
        }
        else
        {
            if (arr[i + 1] > minmax.max)
            {
                minmax.max = arr[i + 1];
            }

            if (arr[i] < minmax.min)
            {
                minmax.min = arr[i];
            }
        }

        i += 2;
    }

    return minmax;
}

int main()
{
    int arr[] = {2 , 7 , 9 , 3 , 4 , 5 , 2 , 1};
    int arr_size = 8;
```

```
    Pair minmax = getMinMax(arr, arr_size);

    cout << "Minimum element is "
         << minmax.min << "\n";
    cout << "Maximum element is "
         << minmax.max;

    return 0;
}
```

Output:

```
Minimum element is 1
Maximum element is 9
Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.
```

Conclusion:

The implementation effectively demonstrates how the divide and conquer technique can optimize the process of identifying the smallest and largest elements in an array. By systematically evaluating pairs and narrowing comparisons, the algorithm achieves a lower time complexity than straightforward iteration. This exercise reinforces the practical value of algorithmic thinking in solving basic computational problems efficiently.

Task 03: A C++ program that will implement the Counting Sort Algorithm.

Theory:

Counting Sort is an efficient, non-comparison-based sorting algorithm particularly useful for sorting integers within a known, limited range. Instead of comparing elements directly like in other sorting algorithms, Counting Sort leverages a counting mechanism to determine the position of each element in the final sorted array.

This technique works in linear time, i.e., O(n + k), where *n* is the number of elements in the input array, and *k* is the range of input values. As it avoids element-wise swaps, it is stable and especially effective for large volumes of data with smaller value ranges.

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.

Code:

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

void countingSort(vector<int>& arr)
{
   if(arr.empty())
   {
      return;
   }

   int maxVal = *max_element(arr.begin(), arr.end());

   vector<int> count(maxVal + 1 , 0); // C
   vector<int> output(arr.size());  // B

   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]; // B[C[A[i]] = A[i]
        count[arr[i]]--; //C[A[i]] = C[A[i]] - 1
    }

    arr = output;
}

int main()
{
    vector<int> data = {4, 2, 2, 8, 3, 3, 1}; //A

    countingSort(data);

    for(int num : data)
    {
        cout << num << " ";
    }

    return 0;
}
```

Output:

```
1 2 2 3 3 4 8
Process returned 0 (0x0)   execution time : 0.110 s
Press any key to continue.
```

Conclusion:

The Counting Sort algorithm efficiently organizes data by leveraging the frequency of input values rather than comparisons. The implementation highlights its advantages in terms of speed and stability, especially when dealing with non-negative integers in a known range. Its simplicity and performance make it a go-to solution in cases where conventional sorting methods fall short due to time complexity constraints.