

Reg .No : 24BCE0554
Name : Partha Pratim Gogoi
Topic : Merge Sort

Algorithm:

```
1 array = [ ... ]                                     [T.C: O(1)]
2 mergeSort(arr)                                     [T.C: O(nlog(n))]
2.1 if (arr.size() <= 1) then return
2.2 split(arr, 0, arr.size()-1)                      [T.C: O(nlog(n))]
3 split(arr, left, right)                           [T.C: O(nlog(n))]
3.1 if (left >= right) then return
3.2 int mid = left + (right-left)/2
3.3 split(arr, left, mid)                          [T.C: T(n)=2T(n/2)+O(n)]
3.4 split(arr, mid+1, right)                      [T.C: T(n)=2T(n/2)+O(n)]
3.5 merge(arr, left, mid, right)                  [T.C: O(n)]
4 merge(arr, left, mid, right)                     [T.C: O(n1+n2)=O(n)]
4.1 n1 = mid - left + 1
4.2 n2 = right - mid
4.3 L[n1] = []
4.4 R[n2] = []
4.5 for i in 0->n1                                [T.C: O(n1)]
    4.5.1 L[i] = arr[left+i]
4.6 for j in 0->n2                                [T.C: O(n2)]
    4.6.1 R[j] = arr[right+j]
4.7 i=0, j=0, k=left
4.8 while (i<n1 && j<n2)                         [T.C: O(n1)|O(n2)]
    4.8.1 if (L[i] <= R[j]) then arr[k++] = L[i++]
    4.8.2 else arr[k++] = R[j++]
4.9 while (i<n1)                                    [T.C: O(n1)]
    4.9.1 arr[k++] = L[i++]
4.10 while (j<n2)                                  [T.C: O(n2)]
    4.10.1 arr[k++] = R[j++]
5 mergeSort(array)                                 [T.C: O(nlog(n))]
```

Time Complexity:

```

1. split()
    split(arr, left, mid)      → T(n/2) [implies 'n/2 time']
    split(arr, mid+1, right)   → T(n/2) [implies 'n/2 time']
    merge( ... )               → O(n)

```

Hence,

$$T.C. = 2T(n/2) + O(n)$$

By Master's Theorem:

$$T(n) = O(n^{\log_2 2} \cdot \log^{8+1}(n)) \\ = O(n \log(n))$$

$$\begin{aligned}\text{Total Time Complexity} &= O(1) + O(n \log(n)) + O(n) \\ &= O(n \log(n))\end{aligned}$$

Source Code:

```

while (i<n1 && j<n2) {
    if (L[i] <= R[j]) { arr[k++] = L[i++]; }
    else             { arr[k++] = R[j++]; }
}

// -----
// Copy remaining elements //                                // This process will occur only for
one array
// -----
while (i<n1) { arr[k++] = L[i++]; }                      // Case 1: Left sub-array has
elements remaining
while (j<n2) { arr[k++] = R[j++]; }                      // Case 2: Right sub-array has
elements remaining
}

///////////////
/// Propagator ///
///////////////
void split(vector<int>& arr, int left, int right, int level, bool isLeft) {
    if (left >= right) return;

    // -----
    // Display current level and subarray being sorted //
    // -----
    cout << "Level " << level << ": Sorting ";

    if (level == 0)      { cout << "      [ "; }
    else if (isLeft)    { cout << "left half   [ "; }
    else                 { cout << "right half  [ "; }

    for (int i=left; i<=right; i++) {
        cout << arr[i] << " ";
    }
    cout << "]" << endl;

    int mid = left + (right - left)/2;

    // -----
    // Create + Sort sub-arrays //                            // Front Propagation
    // -----
    split(arr, left, mid, level+1, true);                  // Left sub-array
    split(arr, mid+1, right, level+1, false);               // Right sub-array

    // -----
    // Merge sub-arrays //                                    // Back Propagation
    // -----
    merge(arr, left, mid, right);
}

///////////////
/// Initiator ///
/////////////
void mergeSort(vector<int>& arr) {
    if (arr.size() <= 1) return;

    split(arr, 0, arr.size()-1, 0, true);                  // Either true/false works during
initiation

```

```

}

///////////
/// Driver Code ///
///////////
int main() {
    vector<int> array = {81,27,56,98,13, 47,26,3,95,78, 26,4,57,23,52, 8,10,23,96,47,
0};

    mergeSort(array);

    cout << endl << "Final sorted array      [ ";
    for (int val: array) {
        cout << val << " ";
    }
    cout << "]" << endl;

    return 0;
}

```

Sample Output:

24BCE0554

```

← 0s @ ./'Merge Sort'/a.out
Level 0: Sorting           [ 81 27 56 98 13 47 26 3 95 78 26 4 57 23 52 8 10 23 96 47 0 ]
Level 1: Sorting left half [ 81 27 56 98 13 47 26 3 95 78 26 ]
Level 2: Sorting left half [ 81 27 56 98 13 47 ]
Level 3: Sorting left half [ 81 27 56 ]
Level 4: Sorting left half [ 81 27 ]
Level 3: Sorting right half [ 98 13 47 ]
Level 4: Sorting left half [ 98 13 ]
Level 2: Sorting right half [ 26 3 95 78 26 ]
Level 3: Sorting left half [ 26 3 95 ]
Level 4: Sorting left half [ 26 3 ]
Level 3: Sorting right half [ 78 26 ]
Level 1: Sorting right half [ 4 57 23 52 8 10 23 96 47 0 ]
Level 2: Sorting left half [ 4 57 23 52 8 ]
Level 3: Sorting left half [ 4 57 23 ]
Level 4: Sorting left half [ 4 57 ]
Level 3: Sorting right half [ 52 8 ]
Level 2: Sorting right half [ 10 23 96 47 0 ]
Level 3: Sorting left half [ 10 23 96 ]
Level 4: Sorting left half [ 10 23 ]
Level 3: Sorting right half [ 47 0 ]

Final sorted array      [ 0 3 4 8 10 13 23 23 26 26 27 47 47 52 56 57 78 81 95 96 98 ]

```

Reg .No : 24BCE0554
Name : Partha Pratim Gogoi
Topic : Huffman Coding

Algorithm:

```
1 text = "message" [T.C: O(1)]
2 freq = [] [T.C: O(1)]
3 struct Node [T.C: O(1)]
4   3.1 char data
5   3.2 unsigned freq
6   3.3 Node *left, *right
7   3.4 Node(data, freq)
8     3.4.1 this->data = data
9     3.4.2 this->freq = freq
10    3.4.3 left = right = NULL
11
12 4 calcFrequencies(string text) [T.C: O(n)]
13    4.1 for char in text O(n)
14      4.2 freq[char]++
15
16 5 buildHuffmanTree(unordered_map(char, int) freq) [T.C: O(n+mlogm)]
17    5.1 priority_queue(Node) minHeap = [] O(1)
18    5.2 for char,freqValue in freq O(m)
19      5.3   minHeap.push(new Node(char, freqValue)) O(log(m))
20    5.4 while minHeap.size > 1 O(m-1)
21      5.5   Node left = minHeap.top()
22      5.6   minHeap.pop() O(log(m))
23      5.7   Node right = minHeap.top()
24      5.8   minHeap.pop() O(log(m))
25      5.9   Node internal = new Node('$',left.freq + right.freq)
26      5.10  internal.left = left
27      5.11  internal.right = right
28      5.12  minHeap.push(internal) O(log(m))
29
30 5.13 return minHeap.top()
31
32 6 storeCodes(
33   Node rootNode,
34   string code,
35   unordered_map(char, int) codes
36 )
37 [T.C: O(m*L)]
38
39 6.1 if rootNode == NULL
40 6.2   return
41 6.3 if rootNode.left == NULL and rootNode.right == NULL
42 6.4   codes[rootNode.data] = code
43 6.5 storeCodes(rootNode.left, code+"0", codes) O(L)
44 6.6 storeCodes(rootNode.right, code+"1", codes) O(L)
```

Time Complexity:

IMPORTANT CLARIFICATION

- A Balanced tree can have $m \neq n$ OR $m = n$
 - 'abcdefg' → balanced tree w/ $m = n$ (special case, all unique)
 - 'aaaabbbbcccc' → balanced tree with $m \neq n$
- A Skewed tree can have $m \neq n$
 - 'bbbbbcda' → generates skewed tree with $m \neq n$
- It is not possible to get a Skewed tree with $m = n$

1. buildHuffmanTree()

A temporary data structure 'minHeap' used to create the Huffman Tree.

Time Complexity:

$$O(m\log(m)) + O((m-1)*3\log(m)) = O(m\log(m))$$

where m = no. of unique characters

- Repeated chars ($m \neq n$) = $O(m\log(m))$
 - Same for skewed and balanced tree
- When all characters unique ($m = n$) = $O(n\log(n))$

2. storeCodes()

- Total nodes visited : $2m - 1$ ' m ' original + ' $m-1$ ' internal
- At each node : $O(L)$ String concatenation
- $m \neq n$ Case $\rightarrow O((2m-1)*L) : O(m*L)$ L = Huffman Tree Height
 - Balanced Tree $\rightarrow O(m\log(m))$
 - Skewed Tree $\rightarrow O(m^2)$
- $m = n$ Case $\rightarrow O((2n-1)*L) : O(n*L)$

Where,

Huffman Tree Height (L) is determined by
no. of unique characters (m)

NOTE: calcFrequency() is $O(n)$ and not dependent on ' m ' or ' L '

So, including calcFrequency() the total time complexity will be:

$$\text{Total Time Complexity} = O(n) + O(m\log(m)) + O(m*L)$$

- Balanced Tree (Typical/Expected Structure):
 $O(n) + O(m\log(m)) + O(m\log(m))$

Total Time Complexity

$$\text{Repeated chars } (m \neq n) = O(n + m\log m)$$

$$\text{All chars unique } (m = n) = O(n\log n) \quad (\text{special case})$$

- Skewed Tree (Pathological Structure):

$$O(n) + O(m\log(m)) + O(m^2)$$

Only one case

$$\text{Repeated chars} = O(m^2)$$

Source Code:

```
#include <iostream>
#include <queue>
#include <unordered_map>
#include <vector>
#include <string>
using namespace std;

struct Node {
    char data;
    unsigned freq;
    Node *left, *right;

    Node(char data, unsigned freq) {
        this->data = data;
        this->freq = freq;
        left = right = nullptr;
    }
};

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

// Non-Recursive Function
Node* buildHuffmanTree(unordered_map<char, int>& freq) {
    // Define minHeap
    priority_queue<Node*, vector<Node*>, Compare> minHeap;

    // Populate minHeap
    for (auto pair: freq) {
        minHeap.push(new Node(pair.first, pair.second));
    }
    // Build Huffman Tree
    while (minHeap.size() > 1){
        // Select the two elements from the minHeap
        // having least frequencies
        Node* left = minHeap.top();
        minHeap.pop();
        Node* right = minHeap.top();
        minHeap.pop();

        // Generate non-leaf node
        // Generated node character -> '$'
        Node* internal = new Node('$', left->freq + right->freq);
        internal->left = left;
        internal->right = right;

        // Push generated node to tree
        minHeap.push(internal);
    }
    return minHeap.top();
}

// Recursive Function
```

```

// Tree is traversed top-to-bottom
void storeCodes(Node* root, string code, unordered_map<char, string>& codes) {
    if (!root) return;
    if (!root->left && !root->right) {
        codes[root->data] = code;
        return;
    } // If at leaf node, character code is complete
    storeCodes(root->left, code+"0", codes);           // If left edge, append 0
    storeCodes(root->right, code+"1", codes);          // If right edge, append 1
}

int main() {
    // 1. Get text input
    string text;
    cout << "Enter text to encode: ";
    getline(cin, text);

    // 2. Calculate frequencies
    cout << endl << "Calculating frequencies..." << endl;
    unordered_map<char, int> freq;
    for (char c: text) {
        freq[c]++;
    }
    // 3. Build the tree
    cout << "Building Huffman tree..." << endl;
    Node* root = buildHuffmanTree(freq);

    // 4. Generate the huffman codes
    cout << "Generating Huffman encoding table..." << endl;
    unordered_map<char, string> codes;
    storeCodes(root, "", codes);
    cout << endl;

    // 4. Print out encoding table
    cout << "Huffman Encoding Table" << endl;
    for (auto pair: codes) {
        cout << pair.first << ": " << pair.second << endl;
    }
    cout << endl;

    // 5. Print out compression results
    int originalBits = 8*text.length();
    int compressedBits = 0;
    string compressedMessage = "";

    for (char c: text) {
        compressedBits += codes[c].length();
        compressedMessage += codes[c];
    } // Calculating new length of compressed message

    cout << "Compressed Message: " << compressedMessage << endl << endl;
    cout << "No. of bits required to store original message: " << originalBits << endl;
    cout << "No. of bits required to store compressed version: " << compressedBits <<
endl;
    cout << "Size reduction: " << (originalBits-compressedBits)*100/originalBits << "%"
<< endl;

    return 0;
}

```

Sample Output:

24BCE0554

```
→ 0s ◊ ./'Huffman Coding'/a.out
Enter text to encode: hello world

Calculating frequencies...
Building Huffman tree...
Generating Huffman encoding table...

Huffman Encoding Table
: 1111
d: 1110
o: 110
l: 10
h: 011
w: 010
r: 001
e: 000
```

□ 25-26/WinSem-Sem3-DAA 16:58

Compressed Message: 0110001010110111010110001101110

No. of bits required to store original message: 88
No. of bits required to store compressed version: 32
Size reduction: 63%

Reg .No : 24BCE0554
Name : Partha Pratim Gogoi
Topic : Fractional Knapsack

Algorithm:

```
1 Capacity = 15 [T.C: O(1)]
2 profit = [ .. ] [T.C: O(1)]
3 weight = [ .. ] [T.C: O(1)]
4 struct Item [T.C: O(1)]
    4.1 int id
    4.2 int profit
    4.3 int weight
    4.4 double ratio
    4.5 Constructor(id, profit, weight) [T.C: O(1)]
        4.5.1 this->id = id
        4.5.2 this->profit = profit
        4.5.3 this->weight = weight
        4.5.4 this->ratio = profit/weight
5 vector<Item> items [T.C: O(1)]
6 populateItemsIntoArray(profits, weight) [T.C: O(n)]
    6.1 for i in profit [T.C: O(n)]
        6.2 push_back(Item(i, profit[i], weight[i])) [T.C: O(1)]
7 compare(a, b) [T.C: O(1)]
    7.1 return a.profit > b.profit
8 fractionalKnapsack(capacity, items) [T.C: O(n+nlog(n))]
    8.1 sort(items[0], items[-1], compare) [T.C: O(nlog(n))]
    8.2 totalProfit = 0.0
    8.3 currentWeight = 0
    8.4 for item in items [T.C: O(n)]
        8.4.1 if currentWeight + item.weight<=capacity [T.C: O(1)]
            8.4.1.1 currentWeight += item.weight
            8.4.1.2 totalProfit += item.profit
        8.4.2 else [T.C: O(1)]
            8.4.2.1 remaining = capacity-currentWeight
            8.4.2.2 if remaining>0 [T.C: O(1)]
                8.4.2.2.1 fraction = remaining/item.weight
                8.4.2.2.2 fractionalProfit = item.profit*fraction
            8.4.2.3 break
    8.5 return totalProfit [T.C: O(1)]
```

Time Complexity:

1. `sort(items[0], items[-1], compare)`

This function has been implemented in "algorithm" library of C++ to have $O(n \log n)$ time complexity

$$\begin{aligned}\text{Total Time Complexity} &= O(1) + O(n) + O(n \log(n)) \\ &= O(n + n \log(n)) \\ &= O(n \log(n))\end{aligned}$$

Source Code:

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

///////////
/// Define item ///
///////////
struct Item {
    int id;
    int profit;
    int weight;
    double ratio;

    // -----
    // Constructor //
    // -----
    Item(int id, int profit, int weight) {
        this->id = id;
        this->profit = profit;
        this->weight = weight;
        ratio = (double) profit/weight;
    }
};

///////////
/// Sort Logic ///
///////////
bool compare(Item a, Item b) {
    // -----
    // Greedy Choice:- Highest P:W Ratio //
    // -----
    return a.ratio > b.ratio;
}

///////////
/// Main Logic ///
///////////
double fractionalKnapsack(int M, vector<Item>& items) {
    // -----
    // Sort highest to lowest P:W ratio //
    // -----
    cout << "Sorting 'items' [highest to lowest P:W]..." << endl;
    sort(items.begin(), items.end(), compare);
```

```

double totalProfit = 0.0;
int currentWeight = 0;

// -----
// Make Greedy Choices //
// -----
cout << "Making greedy choices..." << endl;
for (auto& item: items) {
    if (currentWeight + item.weight <= M) {

        // Incrementing values
        currentWeight += item.weight;
        totalProfit += item.profit;
    } else {
        cout << "Taking current highest P:W item exceeds capacity. Taking fractional
part..." << endl;
        // This case is when we have to take a fractional
        // part an item to reach max profit
        int remaining = M - currentWeight;
        if (remaining > 0) {
            double fraction = (double)remaining / item.weight;
            double fractionalProfit = item.profit * fraction;

            totalProfit += fractionalProfit;
        }
        // Break since capacity is filled
        break;
    }
}
// -----
// Return maximum possible profit //
// -----
return totalProfit;
}

///////////////
/// Driver Code ///
/////////////
int main() {
    cout << "Initializing starting conditions..." << endl;
    int n=10;      // No. of objects
    int M=15;      // Knapsack capacity
    int profit[] = {10, 5, 15, 7, 6, 18, 3, 12, 20, 8};
    int weight[] = {2, 3, 5, 7, 1, 4, 1, 6, 5, 2};

    // -----
    // Populating 'items' array //
    // -----
    cout << "Populating 'items' array..." << endl;
    vector<Item> items;
    for (int i=0; i<n; i++) {
        items.push_back(Item(i, profit[i], weight[i]));
    }
    // -----
    // Apply fractional knapsack logic //
    // -----
    cout << "Applying Fractional Knapsack..." << endl;
    double maxProfit = fractionalKnapsack(M, items);
}

```

```
    cout << endl << "Maximum Profit: " << fixed << setprecision(2) << maxProfit <<
endl;

    return 0;
}
```

Sample Output:

```
24BCE0554

↳ 0s ◎ ./'Fractional Knapsack'/a.out          □ 25-26/WinSem-Sem3-DAA 16:58
Initializing starting conditions...
Populating 'items' array...
Applying Fractional Knapsack...
Sorting 'items' [highest to lowest P:W]...
Making greedy choices...
Taking current highest P:W item exceeds capacity. Taking fractional part...

Maximum Profit: 65.00
```