Name: Vivek Vikram Pundkar

Roll no: 77

GR no: 11910860

Division: C

Batch: 3

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Fractional Knapsack Problem**

**Problem Statement**

Given a set of N items each having value V with weight W and the total capacity of a knapsack. The task is to find the maximal value of fractions of items that can fit into the knapsack.

Examples:

Input: A[] = {{60, 20} , {100, 50}, {120, 30}}, Total\_capacity = 50

Output: 180.00

Explanation: Take the first item and the third item. Total value = 60 + 120 = 180 with a total capacity of 20 + 30 = 50

Input: A[] = {{500, 30}}, Total\_capacity = 10

Output: 166.67

Explanation: Since the total capacity of the knapsack is 10, consider one-third of the item.

**Brute Force Approach**

The most basic approach is to try all possible subsets and possible fractions of the given set and find the maximum value among all such fractions.

The time complexity will be exponential, as you need to find all possible combinations of the given set.

**Greedy Approach**

The Fractional Knapsack problem can be solved efficiently using the greedy algorithm, where you need to sort the items according to their value/weight ratio.

**Algorithm**

* Sort the given array of items according to weight / value(W /V) ratio in descending order.
* Start adding the item with the maximum W / V ratio.
* Add the whole item, if the current weight is less than the capacity, else, add a portion of the item to the knapsack.
* Stop, when all the items have been considered and the total weight becomes equal to the weight of the given knapsack.

**Pseudo Code:**

struct item

{

    int value,weight;

};

bool cmp(item a,item b)

{

    double r1=(double)a.value/a.weight;

    double r2=(double)b.value/b.weight;

    return(r1>r2);

}

double fractionalknapsack(item A[],int Total\_Capacity,int n)

{

    sort(A,A + n,cmp);

    int cur\_weight = 0;

    double final\_val = 0.0;

    for(int i=0;i<n;i++)

    {

        if(cur\_weight + arr[i].weight <= Total\_Capacity)

        {

            Cur\_weight += A[i].weight;

            Final\_val += A[i].value;

        }

        else

        {

            int remain = Total\_capacity - cur\_weight;

            Final\_val += A[i].value \* ((double)remain / A[i].weight);

        }

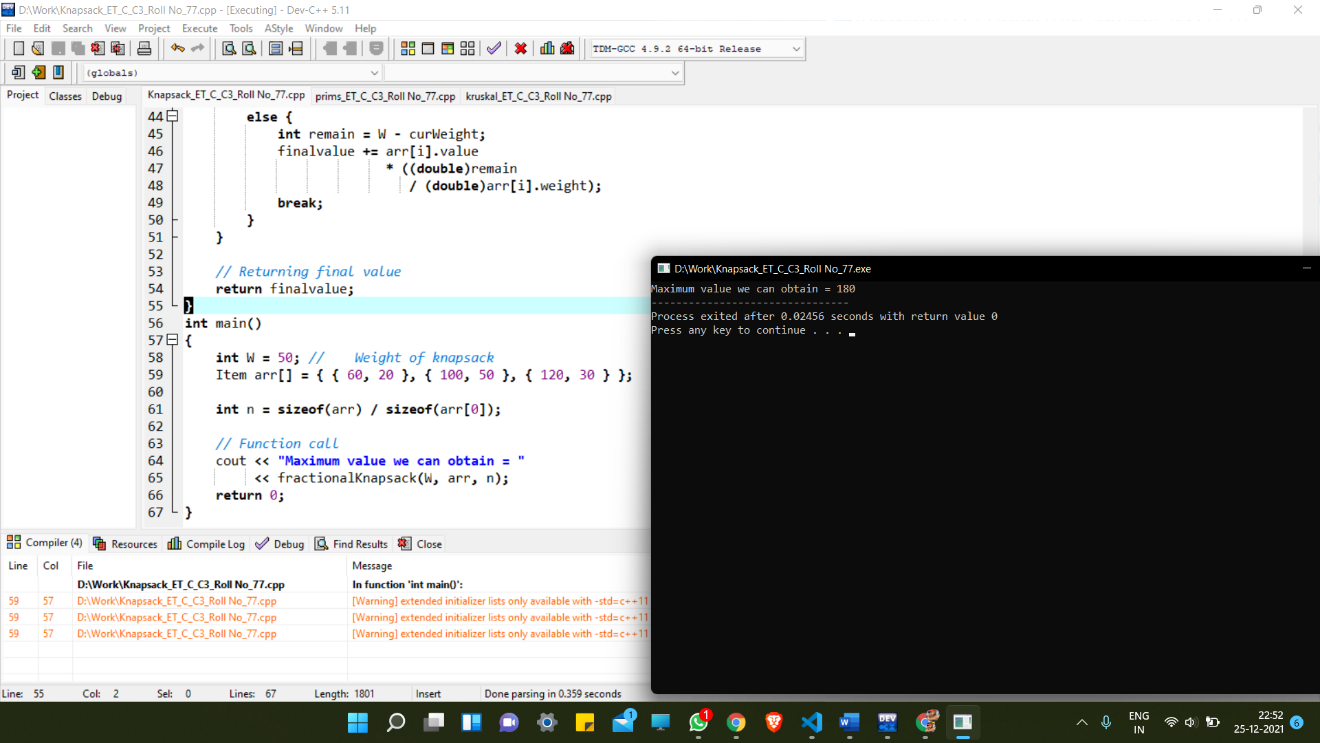
    }

    return final\_val;

}

**Code:**

**Output:**



**Analysis:**

Time Complexity: O(N \*log N) where N is the size of the array.

Space Complexity: O(1) because no extra space is needed.

**Huffman Coding Using Greedy**

**Algorithm:**

1. There are mainly two major parts in Huffman Coding
2. Build a Huffman Tree from input characters.
3. Traverse the Huffman Tree and assign codes to characters.
4. Steps to build Huffman Tree
5. Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.
6. 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)
7. Extract two nodes with the minimum frequency from the min heap.
8. 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.
9. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

**Code:**

#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();

        top = new MinHeapNode('$', left->freq + right->freq);

        top->left = left;

        top->right = right;

        minHeap.push(top);

    }

    printCodes(minHeap.top(), "");

}

int main()

{

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

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

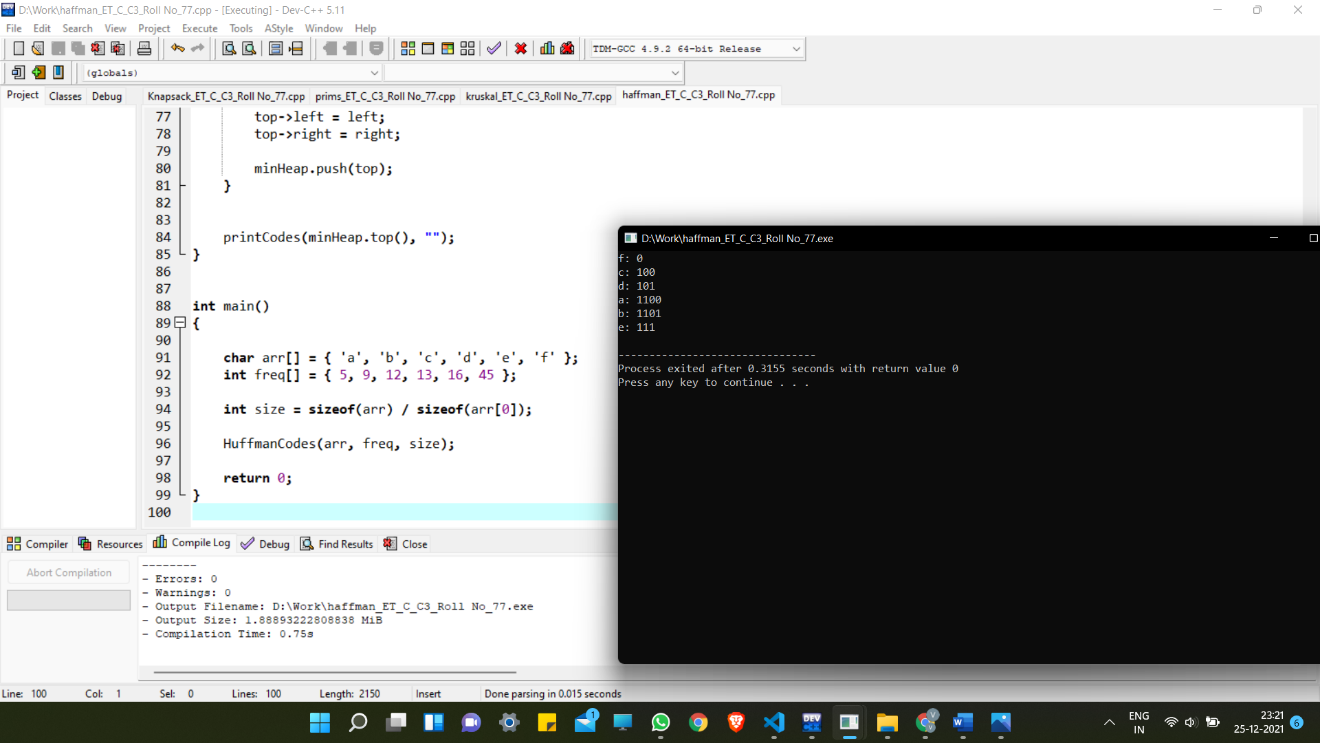
    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;

}

**Output:**



**Time Complexity Analysis:**

1. O(nlogn) where n is the number of unique characters.
2. If there are n nodes, extractMin() is called 2\*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify().
3. So, overall complexity is **O(nlogn).**