**ET3272: Design and Analysis of Algorithms**

**Name of the student: Aditya Jannawar Roll No.: 02**

**Div.: A Batch: 1**

**Date of performance: /0 /2023**

**Experiment No. 13**

# Title: Fractional Knapsack Problem

**Theory/Description of the Problem Statement:**

The Fractional Knapsack Problem is a classic optimization problem in computer science and operations research. It deals with finding the optimal way of filling a knapsack of fixed capacity with items of different weights and values, where fractions of items are allowed. Each item has a weight and a value, and the goal is to maximize the total value of the items that can be carried in the knapsack.

The problem can be formulated as follows: given n items, each with a weight w\_i and a value v\_i, and a knapsack with a capacity W, the goal is to choose items that can be placed in the knapsack in such a way as to maximize the total value of the items while not exceeding the knapsack's capacity.

The Fractional Knapsack Problem can be solved using a greedy algorithm, where we sort the items by their value per unit weight (v\_i / w\_i) in descending order and fill the knapsack with as much of the highest value items as possible until it is full. If an item cannot fit entirely into the knapsack, we take a fraction of it proportional to the available space in the knapsack.

The time complexity of the Fractional Knapsack Problem using a greedy algorithm is O(n log n), where n is the number of items. This is because we need to sort the items by their value per unit weight, which takes O(n log n) time. The actual filling of the knapsack takes linear time O(n).

**Algorithm :**

* Compute the value per unit weight for each item in the list.
* Sort the items in descending order based on their value per unit weight.
* Set the total weight of the knapsack to zero.
* For each item in the sorted list: a. If the item fits entirely in the knapsack, add its total weight and value to the knapsack. b. If the item does not fit entirely in the knapsack, add a fraction of the item to the knapsack such that the total weight of the knapsack is not exceeded.
* Return the total value of the items in the knapsack.

**Pseudo Code :**

* // Structure for an item which stores weight and corresponding value of Item
* struct Item {
* int profit, weight;
* // Constructor
* Item(int profit, int weight) {
* this->profit = profit;
* this->weight = weight;
* }
* };
* // Comparison function to sort Item according to profit/weight ratio
* static bool cmp(struct Item a, struct Item b) {
* double r1 = (double)a.profit / (double)a.weight;
* double r2 = (double)b.profit / (double)b.weight;
* return r1 > r2;
* }
* // Main greedy function to solve problem
* double fractionalKnapsack(int W, struct Item arr[], int N) {
* // Sorting Item on basis of ratio
* sort(arr, arr + N, cmp);
* double finalvalue = 0.0;
* // Looping through all items
* for (int i = 0; i < N; i++) {
* // If adding Item won't overflow, add it completely
* if (arr[i].weight <= W) {
* W -= arr[i].weight;
* finalvalue += arr[i].profit;
* }
* // If we can't add current Item, add fractional part of it
* else {
* finalvalue += arr[i].profit \* ((double)W / (double)arr[i].weight);
* break;
* }
* }
* // Returning final value
* return finalvalue;
* }

**Analysis of the Algorithm**

**Time Complexity:**

Sorting the items by their profit/weight ratio takes O(n log n) time. The for loop that iterates over the items takes O(n) time.Each item requires constant time operations to check if it can be added to the knapsack and to add its value to the final value, so the total time complexity for the loop is O(n).Therefore, the overall time complexity of the algorithm is O(n log n), dominated by the sorting step.

**Space Complexity:**

The only additional data structure used in the algorithm is the array of Item structs, which takes O(n) space. Therefore, the overall space complexity of the algorithm is O(n).

**Experiment and result:**

Code:

// C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>

using namespace std;

// Structure for an item which stores weight and

// corresponding value of Item

struct Item {

    int profit, weight;

    // Constructor

    Item(int profit, int weight)

    {

        this->profit = profit;

        this->weight = weight;

    }

};

// Comparison function to sort Item

// according to profit/weight ratio

static bool cmp(struct Item a, struct Item b)

{

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

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

    return r1 > r2;

}

// Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int N)

{

    // Sorting Item on basis of ratio

    sort(arr, arr + N, cmp);

    double finalvalue = 0.0;

    // Looping through all items

    for (int i = 0; i < N; i++) {

        // If adding Item won't overflow,

        // add it completely

        if (arr[i].weight <= W) {

            W -= arr[i].weight;

            finalvalue += arr[i].profit;

        }

        // If we can't add current Item,

        // add fractional part of it

        else {

            finalvalue

                += arr[i].profit

                \* ((double)W / (double)arr[i].weight);

            break;

        }

    }

    // Returning final value

    return finalvalue;

}

// Driver code

int main()

{

    int W = 50;

    Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

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

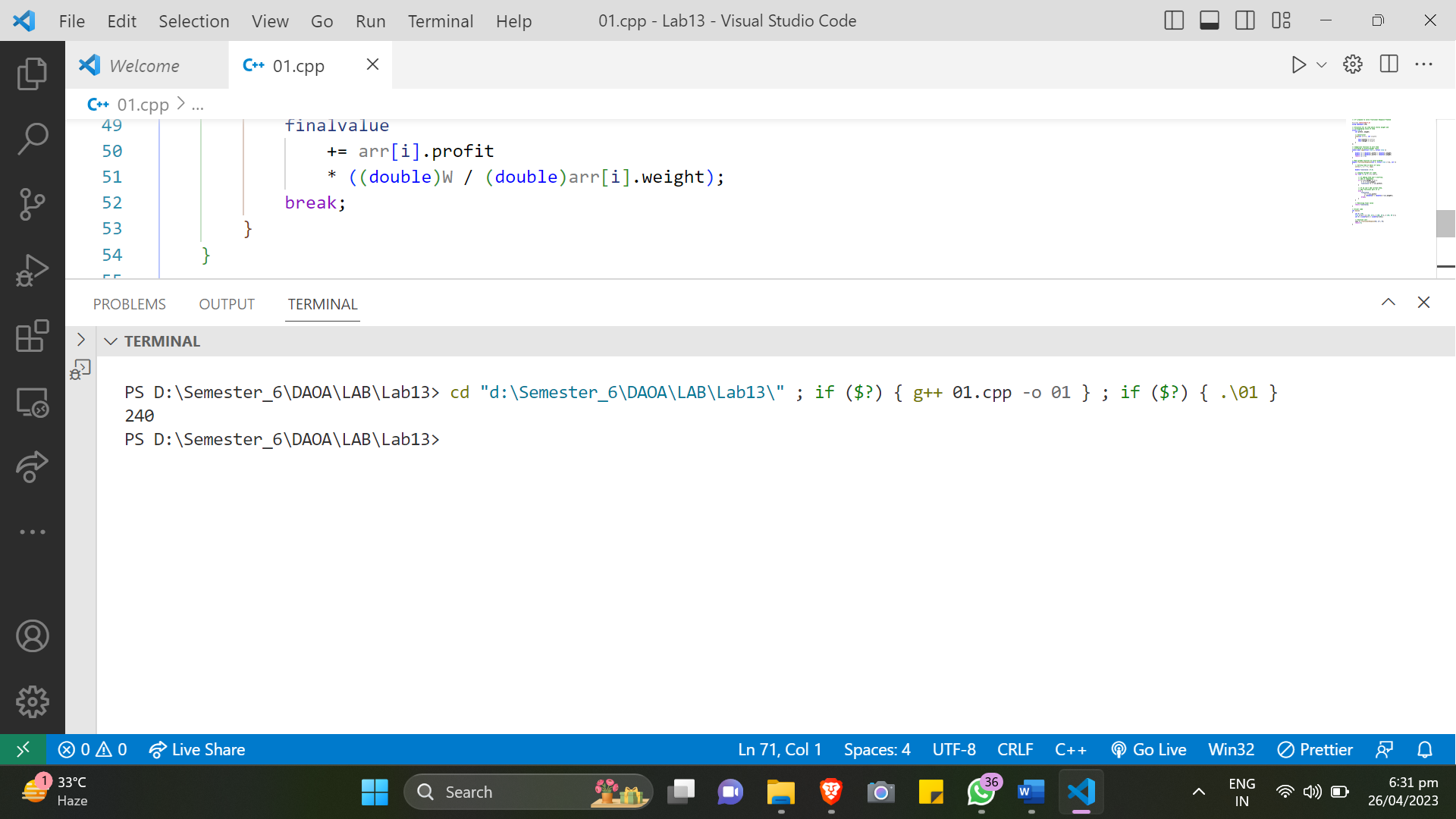
    // Function call

    cout << fractionalKnapsack(W, arr, N);

    return 0;

}

Output:



**Conclusions:**

The Fractional Knapsack problem is a popular algorithmic problem in computer science and optimization. It involves selecting items with specific weights and values to be included in a knapsack, with the objective of maximizing the value of the items in the knapsack while adhering to its weight capacity. The greedy approach to this problem is to sort the items by their value/weight ratio and add as much of each item as possible until the knapsack's capacity is reached. The time complexity of this algorithm is O(nlogn), where n is the number of items, and the space complexity is O(1) because the algorithm does not require additional memory beyond the input items. This algorithm is often used as a building block for other optimization problems, and it has real-world applications in resource allocation, portfolio optimization, and other areas.