ALGORITHM LABORATORY

ASSIGNMENT-6

PROBLEM STATEMENT: Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

ALGORITHM (Fractional Knapsack):

a. Input Handling
    i. Accept user input for the number of items (n) and maximum weight capacity (W).
    ii. Accept item details (weight and value) or generate random test cases for predefined sizes.
b. Calculate Value-to-Weight Ratio:
    i. For each item, compute its ratio as value/weight.
c. Sort Items:
    i. Sort the items in descending order of their value-to-weight ratio.
d. Maximize Value:
    i. Initialize totalValue = 0 and remainingCapacity = W.
    ii. Traverse the sorted items:
        1. If the item fits entirely, add its full value to totalValue.
        2. If it partially fits, add the fractional value based on the remaining capacity.
e. Display Results:
    i. Print the maximum achievable value.
    ii. Measure and display the execution time for performance analysis.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;
using namespace chrono;

struct Product {
    int mass, worth;
    double worthPerMass;
};

bool sortByRatio(Product a, Product b) {
    return a.worthPerMass > b.worthPerMass;
}


double optimalKnapsack(int capacity, vector<Product> &items, int totalItems, bool
showSteps) {
    sort(items.begin(), items.end(), sortByRatio);

    double maxWorth = 0.0;
    int remainingCapacity = capacity;

    for (int i = 0; i < totalItems && remainingCapacity > 0; i++) {
        if (items[i].mass <= remainingCapacity) {
            maxWorth += items[i].worth;
            remainingCapacity -= items[i].mass;
        } else {
            maxWorth += items[i].worthPerMass * remainingCapacity;
            remainingCapacity = 0;
        }
    }

    if (showSteps) {
        cout << "\nSorted Items (by Value/Weight ratio):\n";
        for (auto item : items)
            cout << "Mass: " << item.mass << ", Worth: " << item.worth
                 << ", Ratio: " << fixed << setprecision(2) << item.worthPerMass << endl;
    }
    return maxWorth;
}

int main() {
    srand(time(0));
    int option;

    cout << "Choose input method:\n";
    cout << "1. Enter items manually\n";
    cout << "2. Generate random items (Test cases: 100, 500, 1000, 5000, 10000)\n";
    cout << "Enter option: ";
```

```
    cin >> option;

    if (option == 1) {
        int totalItems, capacity;
        cout << "Enter number of items: ";
        cin >> totalItems;
        cout << "Enter capacity limit: ";
        cin >> capacity;

        vector<Product> items(totalItems);
        cout << "Enter mass and worth for each item:\n";
        for (int i = 0; i < totalItems; i++) {
            cout << "Item " << i + 1 << " - Mass: ";
            cin >> items[i].mass;
            cout << "        Worth: ";
            cin >> items[i].worth;
            items[i].worthPerMass = (double)items[i].worth / items[i].mass;
        }

        bool showSteps = (totalItems <= 5);
        auto start = high_resolution_clock::now();
        double maxWorth = optimalKnapsack(capacity, items, totalItems, showSteps);
        auto end = high_resolution_clock::now();
        auto execTime = duration_cast<milliseconds>(end - start);

        cout << "\nMaximum Worth in Fractional Knapsack: " << fixed << setprecision(2) <<
maxWorth;
        cout << "\nExecution Time: " << execTime.count() << " ms\n";

    } else if (option == 2) {
        vector<int> testSizes = {100, 500, 1000, 5000, 10000};
        int maxCapacity = 5000;

        for (int totalItems : testSizes) {
            int capacity = rand() % maxCapacity + 1000;

            vector<Product> items(totalItems);
            for (int i = 0; i < totalItems; i++) {
                items[i].mass = rand() % 100 + 1;
                items[i].worth = rand() % 200 + 1;
                items[i].worthPerMass = (double)items[i].worth / items[i].mass;
            }

            cout << "\n====================================";
            cout << "\nTesting for Items = " << totalItems << ", Capacity = " << capacity;
            cout << "\n====================================";

            bool showSteps = (totalItems <= 5);
            auto start = high_resolution_clock::now();
            double maxWorth = optimalKnapsack(capacity, items, totalItems, showSteps);
            auto end = high_resolution_clock::now();
            auto execTime = duration_cast<milliseconds>(end - start);
```

```
            cout << "\nMaximum Worth in Fractional Knapsack: " << fixed << setprecision(2)
<< maxWorth;
            cout << "\nExecution Time: " << execTime.count() << " ms\n";
        }
    } else {
        cout << "Invalid choice! Exiting...\n";
    }
    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(n log n)
    - This happens when the items are already sorted in descending order by their value-to-weight ratio. The sorting step is still O(n log n), but the traversal step efficiently fills the knapsack.
- Average Case: O(n log n)
- Worst Case: O(n log n)
    - Even in the worst scenario, the sorting step determines the upper limit of the complexity. The additional steps (like iterating through the items and calculating the maximum achievable value) are linear.

OUTPUT:

Choose input method:

1. Enter items manually

2. Generate random items (Test cases: 100, 500, 1000, 5000, 10000)

Enter option: 1

Enter number of items: 3

Enter capacity limit: 50

Enter mass and worth for each item:

Item 1 - Mass: 10

Worth: 60

Item 2 - Mass: 20

Worth: 100

Item 3 - Mass: 30

Worth: 120

Sorted Items (by Value/Weight ratio):

Mass: 10, Worth: 60, Ratio: 6.00

Mass: 20, Worth: 100, Ratio: 5.00

Mass: 30, Worth: 120, Ratio: 4.00

Maximum Worth in Fractional Knapsack: 240.00

Execution Time: 2 ms

ALGORITHM (01 Knapsack):

a. Input Reading:
    i. Accept user input for the number of items (n) and the maximum weight limit (W).
    ii. Collect item weights and values manually or generate random test cases.
b. Initialize DP Table:
    i. Create a 2D DP table dp[n + 1][W + 1] and initialize all entries to zero.
c. Fill DP Table:
    i. Iterate through items (i = 1 to n).
    ii. For each item, iterate through possible weight capacities (w = 0 to W).
    iii. If the current item's weight is less than or equal to w:
        1. $dp[i][w]=max(values[i-1]+dp[i-1][w-weights[i-1]],dp[i-1][w])$
    iv. Otherwise:
        I. $dp[i][w]=dp[i-1][w]$
d. Trace Back the Selected Items (Optional for Debugging):

i. Start from dp[n][W] and track which items contributed to the optimal value.

e. Output:

i. Display the maximum achievable value in the knapsack.

ii. Display intermediate DP table steps if required.

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;
using namespace chrono;

int computeMaxValue(int capacity, vector<int> &wt, vector<int> &val, int itemCount, bool
showSteps) {
    vector<vector<int>> table(itemCount + 1, vector<int>(capacity + 1, 0));

    for (int idx = 1; idx <= itemCount; idx++) {
        for (int cap = 0; cap <= capacity; cap++) {
            if (wt[idx - 1] <= cap)
                table[idx][cap] = max(val[idx - 1] + table[idx - 1][cap - wt[idx - 1]],
table[idx - 1][cap]);
            else
                table[idx][cap] = table[idx - 1][cap];
        }
    }

    if (showSteps) {
        cout << "\nDP Table (Intermediate Steps):\n";
        for (int i = 0; i <= itemCount; i++) {
            for (int w = 0; w <= capacity; w++) {
                cout << setw(4) << table[i][w] << " ";
            }
            cout << endl;
        }

        cout << "\nItems Selected:\n";
        int cap = capacity;
        for (int idx = itemCount; idx > 0 && cap > 0; idx--) {
            if (table[idx][cap] != table[idx - 1][cap]) {
                cout << "Item " << idx << " (Weight: " << wt[idx - 1] << ", Value: " <<
val[idx - 1] << ")\n";
                cap -= wt[idx - 1];
            }
        }
    }

    return table[itemCount][capacity];
}

int main() {
```

```
    srand(time(0));
    int userChoice;

    cout << "Choose input method:\n";
    cout << "1. Manual item entry\n";
    cout << "2. Auto-generate random items (Test sizes: 100, 500, 1000, 5000, 10000)\n";
    cout << "Enter choice: ";
    cin >> userChoice;

    if (userChoice == 1) {
        int itemCount, maxCapacity;
        cout << "Enter number of items: ";
        cin >> itemCount;
        cout << "Enter maximum weight limit: ";
        cin >> maxCapacity;

        vector<int> wt(itemCount), val(itemCount);
        cout << "Enter weight and value for each item:\n";
        for (int i = 0; i < itemCount; i++) {
            cout << "Item " << i + 1 << " - Weight: ";
            cin >> wt[i];
            cout << "        Value: ";
            cin >> val[i];
        }

        bool showSteps = (itemCount <= 5);
        auto start = high_resolution_clock::now();
        int result = computeMaxValue(maxCapacity, wt, val, itemCount, showSteps);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<milliseconds>(stop - start);

        cout << "\nMaximum Value in 0/1 Knapsack: " << result;
        cout << "\nExecution Time: " << duration.count() << " ms\n";

    } else if (userChoice == 2) {
        vector<int> sampleSizes = {100, 500, 1000, 5000, 10000};
        int maxCapacityLimit = 5000;

        for (int itemCount : sampleSizes) {
            int maxCapacity = rand() % maxCapacityLimit + 1000;

            vector<int> wt(itemCount), val(itemCount);
            for (int i = 0; i < itemCount; i++) {
                wt[i] = rand() % 100 + 1;
                val[i] = rand() % 200 + 1;
            }

            cout << "\n====================================";
            cout << "\nTesting for n = " << itemCount << ", Max Capacity = " <<
maxCapacity;
            cout << "\n====================================";
```

```
            bool showSteps = (itemCount <= 5);

            auto start = high_resolution_clock::now();
            int result = computeMaxValue(maxCapacity, wt, val, itemCount, showSteps);
            auto stop = high_resolution_clock::now();
            auto duration = duration_cast<milliseconds>(stop - start);

            cout << "\nMaximum Value in 0/1 Knapsack: " << result;
            cout << "\nExecution Time: " << duration.count() << " ms\n";
        }
    } else {
        cout << "Invalid choice! Exiting...\n";
    }

    return 0;
}
```

TIME COMPLEXITY:

- Best Case: O(n * W)
  - Occurs when the optimal solution is found early in the DP table's progression (though this scenario is rare since the algorithm fills the entire table).
  - Even in the best case, all entries in the DP table must be filled, making the complexity O(n * W).
- Average Case: O(n * W)
- Worst Case: O(n * W)
  - Occurs when the algorithm processes every item and checks every weight value without any early exits or optimizations.
  - The DP table's size is n * W, and each cell computation requires constant time, resulting in O(n * W).

OUTPUT and PLOT:

Choose input method:

1. Manual item entry

2. Auto-generate random items (Test sizes: 100, 500, 1000, 5000, 10000)

Enter choice: 1

Enter number of items: 3

Enter maximum weight limit: 4

Enter weight and value for each item:

Item 1 - Weight: 4

Value: 1

Item 2 - Weight: 5

Value: 2

Item 3 - Weight: 1

Value: 3

DP Table (Intermediate Steps):

0 0 0 0 0

0 0 0 0 1

0 0 0 0 1

0 3 3 3 3

Items Selected:

Item 3 (Weight: 1, Value: 3)

Maximum Value in 0/1 Knapsack: 3

Execution Time: 4 ms

Node 1 : -91

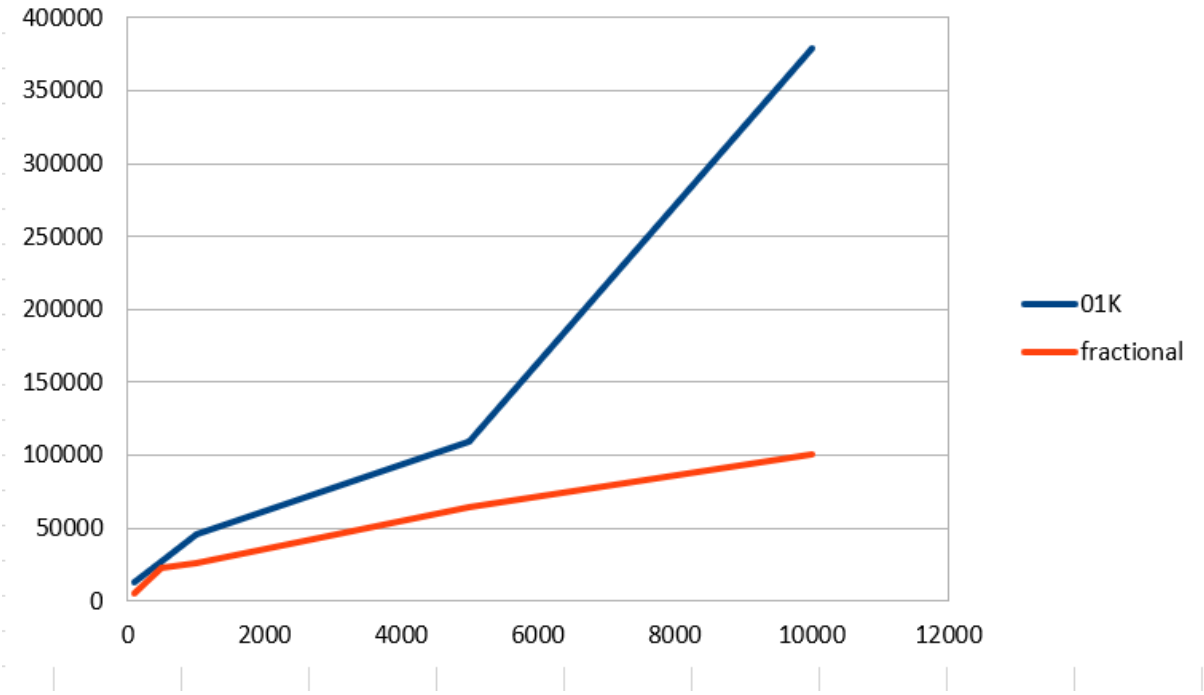
Node 2 : -111

Node 3 : -95

Negative Cycle detected!

Key Conclusion:

- **Use Fractional Knapsack** when precision is less critical, and performance is a priority.
- **Use 0/1 Knapsack** when exact results are required, especially in resource allocation problems where items are indivisible.


400000
350000
300000
250000
200000
150000
100000
50000
0
0
2000
4000
6000
8000
10000
12000
01K
fractional