Problem Statement: Stepwise Execution Analysis of Greedy Algorithms

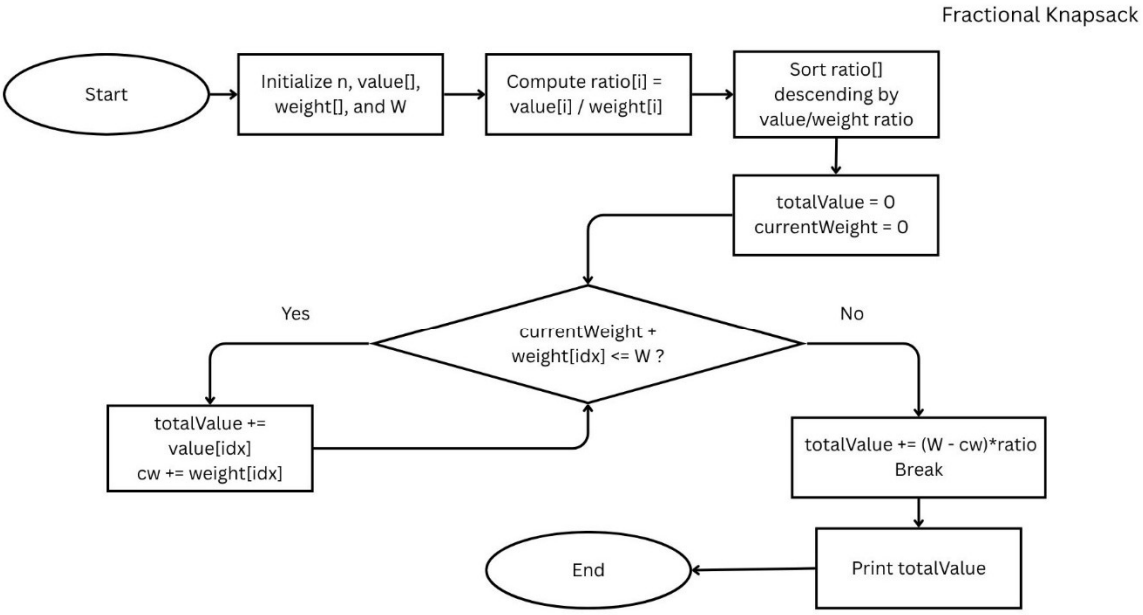
Fractional Knapsack

Theory:

Algorithm:

1. First, find how much value each item gives for one unit of its weight.
2. Arrange all the items in descending order based on this value-per-weight ratio — so the item giving the most value for each unit of weight comes first.
3. Start with an empty knapsack and total value = 0.
4. Go through the sorted list of items one by one:
5. If the current item can fit completely in the knapsack,
   put the whole item in, and subtract its weight from the remaining capacity.
6. If the item cannot fit completely,
   take only the fraction that will exactly fill the remaining space of the knapsack.
   Add that fractional value to the total, and stop (the knapsack is full now).
7. When the knapsack is full or all items are considered,
   the total value you have is the maximum possible value.

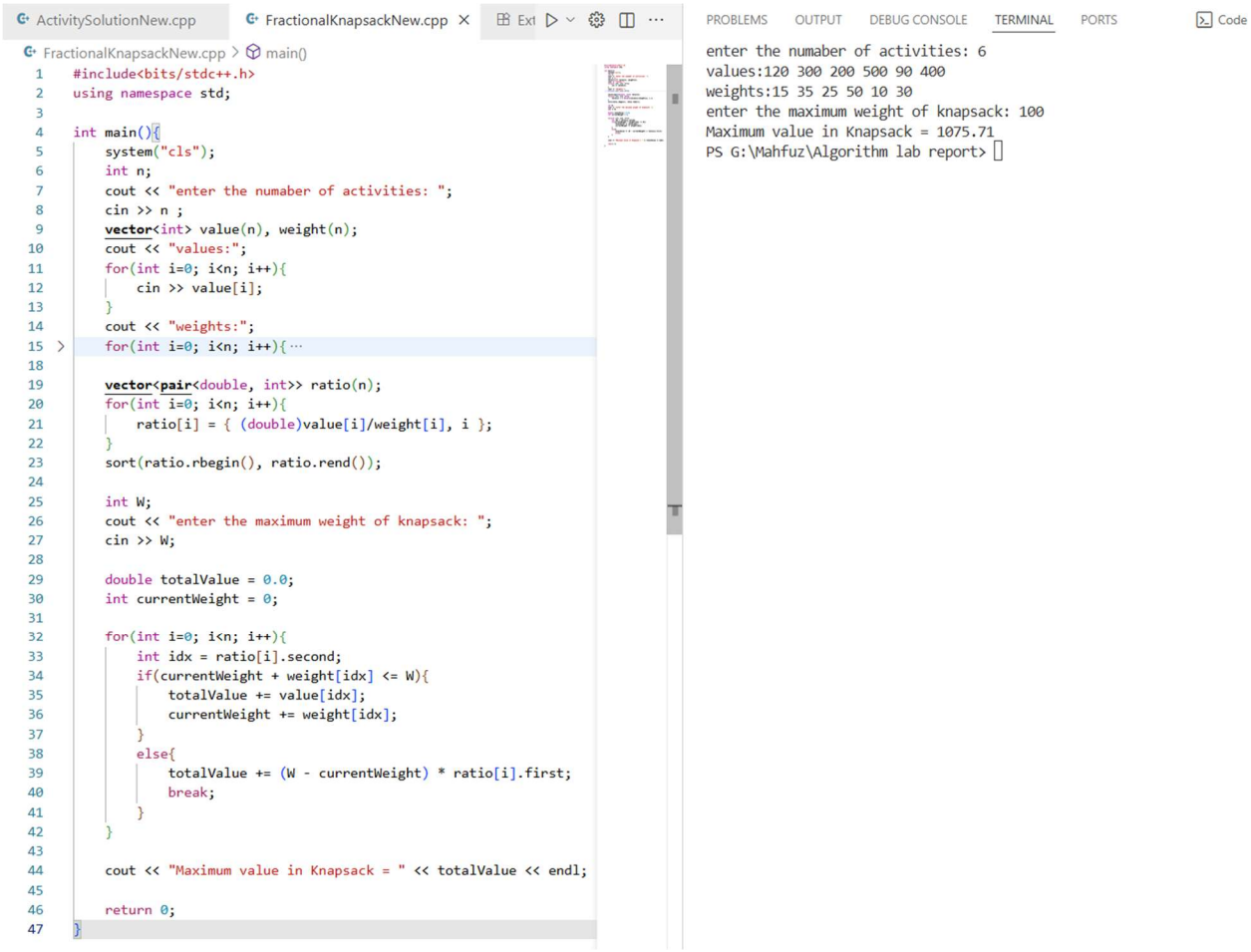
Flowchart:



Code:

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

int main(){
    system("cls");
    int n;
    cout << "enter the numaber of activities: ";
    cin >> n ;
    vector<int> value(n), weight(n);
    cout << "values:";
    for(int i=0; i<n; i++){
        cin >> value[i];
    }
    cout << "weights:";
    for(int i=0; i<n; i++){
        cin >> weight[i];
    }

    vector<pair<double, int>> ratio(n);
    for(int i=0; i<n; i++){
        ratio[i] = { (double)value[i]/weight[i], i };
    }
    sort(ratio.rbegin(), ratio.rend());

    int W;
    cout << "enter the maximum weight of knapsack: ";
    cin >> W;

    double totalValue = 0.0;
    int currentWeight = 0;

    for(int i=0; i<n; i++){
        int idx = ratio[i].second;
        if(currentWeight + weight[idx] <= W){
            totalValue += value[idx];
            currentWeight += weight[idx];
        }
        else{
            totalValue += (W - currentWeight) * ratio[i].first;
            break;
        }
    }

    cout << "Maximum value in Knapsack = " << totalValue << endl;

    return 0;
}
```

Screenshots:



Analysis:

Input Data

Item	Weight (kg)	Value ($)
1	15	120
2	35	300
3	25	200
4	50	500
5	10	90
6	30	400

Step 1: Compute Value/Weight Ratio

Item	Weight	Value	Value/Weight Ratio
1	15	120	8.00
2	35	300	8.57
3	25	200	8.00

4	50	500	10.00
5	10	90	9.00
6	30	400	13.33

Step 2: Sort Items by Ratio (Descending Order)

Rank	Item	Weight	Value	Ratio
1	6	30	400	13.33
2	4	50	500	10.00
3	5	10	90	9.00
4	2	35	300	8.57
5	1	15	120	8.00
6	3	25	200	8.00

Step 3: Take Items Until Knapsack is Full

Let's assume knapsack capacity = 100 kg (you can change this value if given).

Step	Item	Weight Taken	Value Gained	Remaining Capacity
1	6	30	400	70
2	4	50	500	20
3	5	10	90	10
4	2	10 (partial of 35)	(10/35)×300 = 85.7	0

Total Value = 400 + 500 + 90 + 85.7 = 1075.7

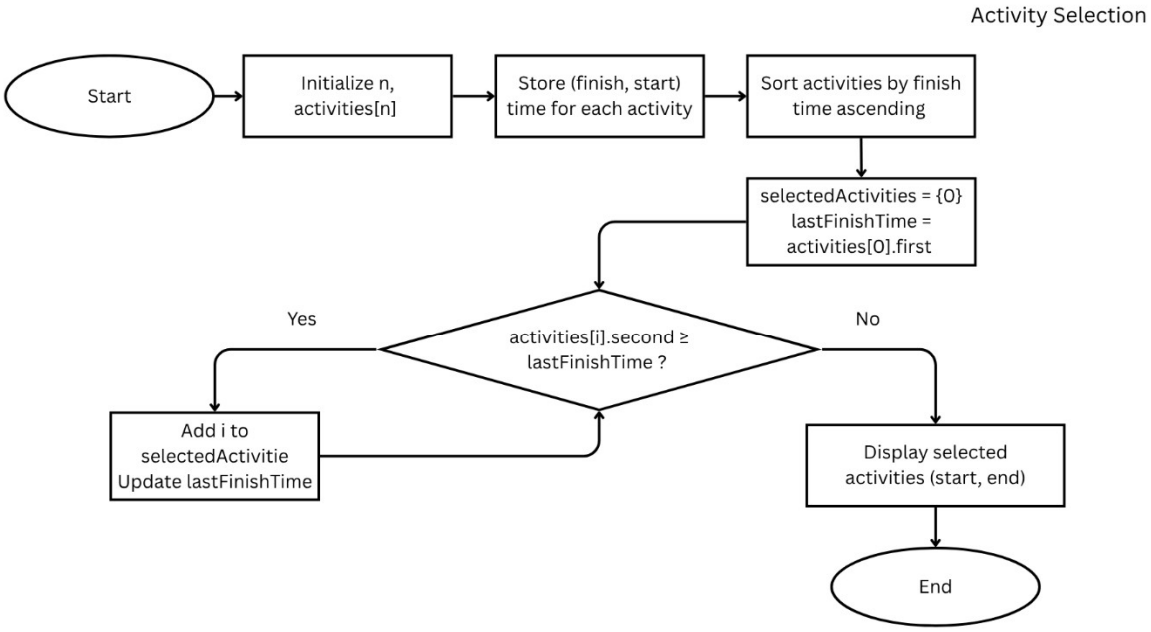
Activity Selection

Theory:

Algorithm:

1. Sort the activities by their finishing times (in ascending order).
2. Select the first activity (which finishes the earliest).
3. Iterate through the remaining activities:
    - If the start time of the current activity is greater than or equal to the finish time of the last selected activity, select it.
4. Continue until all activities are checked.

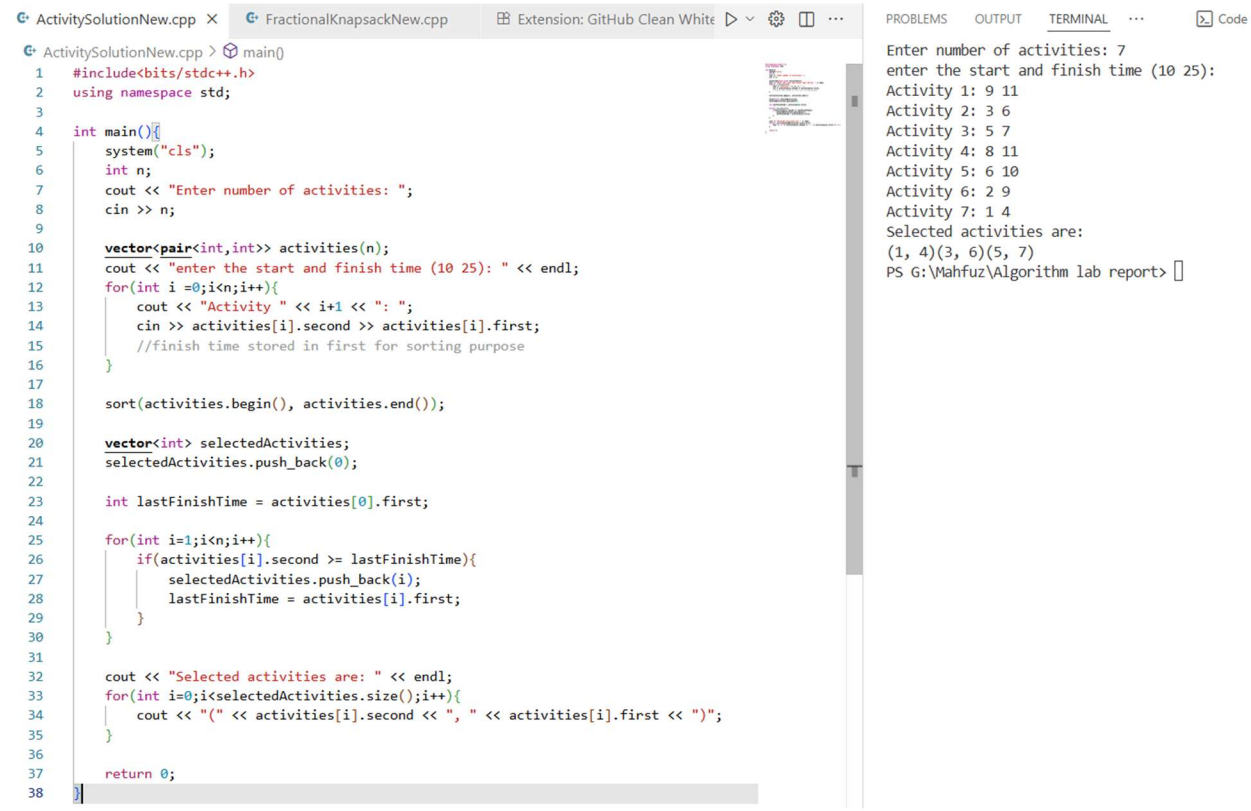
Flowchart:



Code:

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

int main(){
    system("cls");
    int n;
    cout << "Enter number of activities: ";
    cin >> n;

        vector<pair<int,int>> activities(n);
        cout << "enter the start and finish time (10 25): " << endl;
        for(int i =0;i<n;i++){
            cout << "Activity " << i+1 << ": ";
            cin >> activities[i].second >> activities[i].first;
            //finish time stored in first for sorting purpose
        }

        sort(activities.begin(), activities.end());

        vector<int> selectedActivities;
        selectedActivities.push_back(0);

        int lastFinishTime = activities[0].first;

        for(int i=1;i<n;i++){
            if(activities[i].second >= lastFinishTime){
                selectedActivities.push_back(i);
                lastFinishTime = activities[i].first;
            }
        }

        cout << "Selected activities are: " << endl;
        for(int i=0;i<selectedActivities.size();i++){
            cout << "(" << activities[i].second << ", " << activities[i].first << ")";
        }

        return 0;
    }
```

Screenshots:



Analysis:

Input table:

Activity	1	2	3	4	5	6	7
Start	9	3	5	8	6	2	1
Finish	11	6	7	11	10	9	4

Step 1 — Sort activities by finish time

List activities with finish times and sort by finish:

Activity 7 — finish 4 (start 1)

Activity 2 — finish 6 (start 3)

Activity 3 — finish 7 (start 5)

Activity 6 — finish 9 (start 2)

Activity 5 — finish 10 (start 6)

Activity 1 — finish 11 (start 9)

Activity 4 — finish 11 (start 8)

(If two activities tie on finish time, either order is fine — the greedy rule still holds.)

Step 2 — Greedy selection (pass by pass)

Pass 1 — pick the first (earliest finishing):

Choose Activity 7 (1 → 4).

last_finish = 4

Selected: {7}

Pass 2 — scan next activities in sorted order:

Activity 2 starts at 3 → 3 < last_finish(4) → cannot pick (overlaps).

Activity 3 starts at 5 → 5 ≥ 4 → can pick.

Pick Activity 3 (5 → 7).

Update last_finish = 7

Selected: {7, 3}

Pass 3 — continue scanning:

Activity 6 starts at 2 → 2 < 7 → skip.

Activity 5 starts at 6 → 6 < 7 → skip.

Activity 1 starts at 9 → 9 ≥ 7 → can pick.

Pick Activity 1 (9 → 11).

Update last_finish = 11

Selected: {7, 3, 1}

Pass 4 — remaining:

Activity 4 starts at 8 → 8 < 11 → skip.

No more activities that start ≥ last_finish.

Final selected set (greedy result)

Selected activities: 7, 3, 1

Their intervals: (1 → 4), (5 → 7), (9 → 11)

Time and Space Complexity:

Fractional Knapsack Algorithm: The algorithm first calculates the value-to-weight ratio for all n items, taking O(n) time. Then it sorts the items based on this ratio, which requires O(n log n) time. Finally, it iterates through the sorted list once, adding items to the knapsack in O(n) time. Therefore, the overall time complexity is O(n log n). The space complexity is O(n) because the program stores values, weights, and ratio arrays of size n.

Activity Selection Algorithm: This algorithm sorts all n activities according to their finish times in O(n log n) time. After sorting, it scans through the list once to select compatible activities in O(n) time. Thus, the total time complexity is O(n log n). The space complexity is O(n), as it uses arrays (or vectors) to store activity pairs and selected indices. Both algorithms efficiently use greedy strategies for optimization.