SESSIONAL REPORT

Course No: CSE-2202

Experiment No : 06

Name of Experiment:

Introduction to Dynamic Programming Using Knapsack Problem (0/1 and Fractional)

Objectives:

- To understand what Dynamic programming is and why it is used.
- To solve the **0/1 Knapsack problem** using DP.
- To understand the **Fractional Knapsack problem** and solve it using greedy method.
- To compare the 0/1 and fractional knapsack.
- To analyze the time and space complexity for both of the methods.

Introduction:

Dynamic Programming (DP) is a technique for solving problems by breaking the main problem into smaller subproblems and storing their results (memorization) so that we don't need to solve same subproblem again and again. It is useful when a problem has **overlapping subproblems** and **optimal substructure**.

The **Knapsack problem** is one of the most famous DP problems.
We have some items, each of them have a weight and value or profit, and a bag (knapsack) with maximum capacity.
Our target is to select items in such a way so that we can get maximum value in the bag without crossing the weight limit.

There are two types of knapsack problem:

- **0/1 Knapsack Problem:**
  It says each item can either be taken completely (1) or not taken at all (0).(Binary)
  Dynamic Programming is used here because each choice is dependent on previous subproblems.

- **Fractional Knapsack Problem:**
  It says that ,we can take **fractions** of an item.
  This problem is solved using **Greedy method**, not DP.
  We pick items based on value/weight ratio. (from high to low)
  Easier to implement

Both problems need different ideas: DP for 0/1 type choices, and greedy for fractional type.

Algorithm (0/1 Knapsack):

Step 1: Make a DP table with n, w

```
dp[n+1][W+1]
```

where n = number of items and W = capacity.

Step 2: Initialize first row and first column with 0 (no items or no capacity).

Step 3: For each item i and each weight w:

If weight of item i is less than or equal to w:

```
dp[i][w] = max(value[i] + dp[i-1][w - weight[i]], dp[i-1][w])
```

Else:

```
dp[i][w] = dp[i-1][w]
```

Step 4: After filling table, `dp[n][W]` is the required maximum value.

Algorithm (Fractional Knapsack):

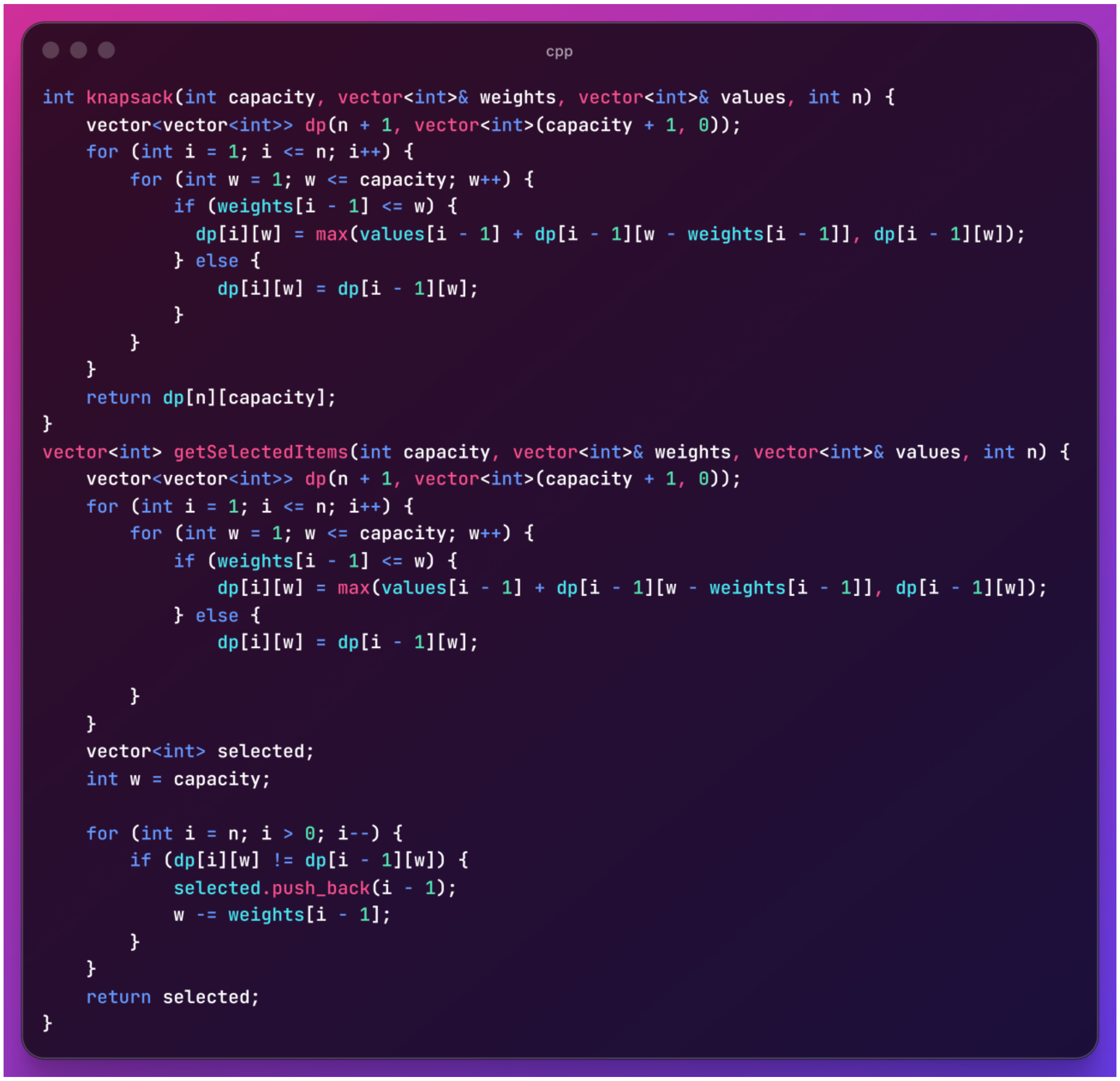
Step 1: Compute value/weight ratio for each item.

Step 2: Sort items in descending order of this ratio.

Step 3: Pick items one by one:
- If item fits completely, take it.
- Otherwise take fraction part that fits in the bag.

Step 4: Stop when the bag/ knapsack is full.

Code for implementation(0/1 Knapsack) :

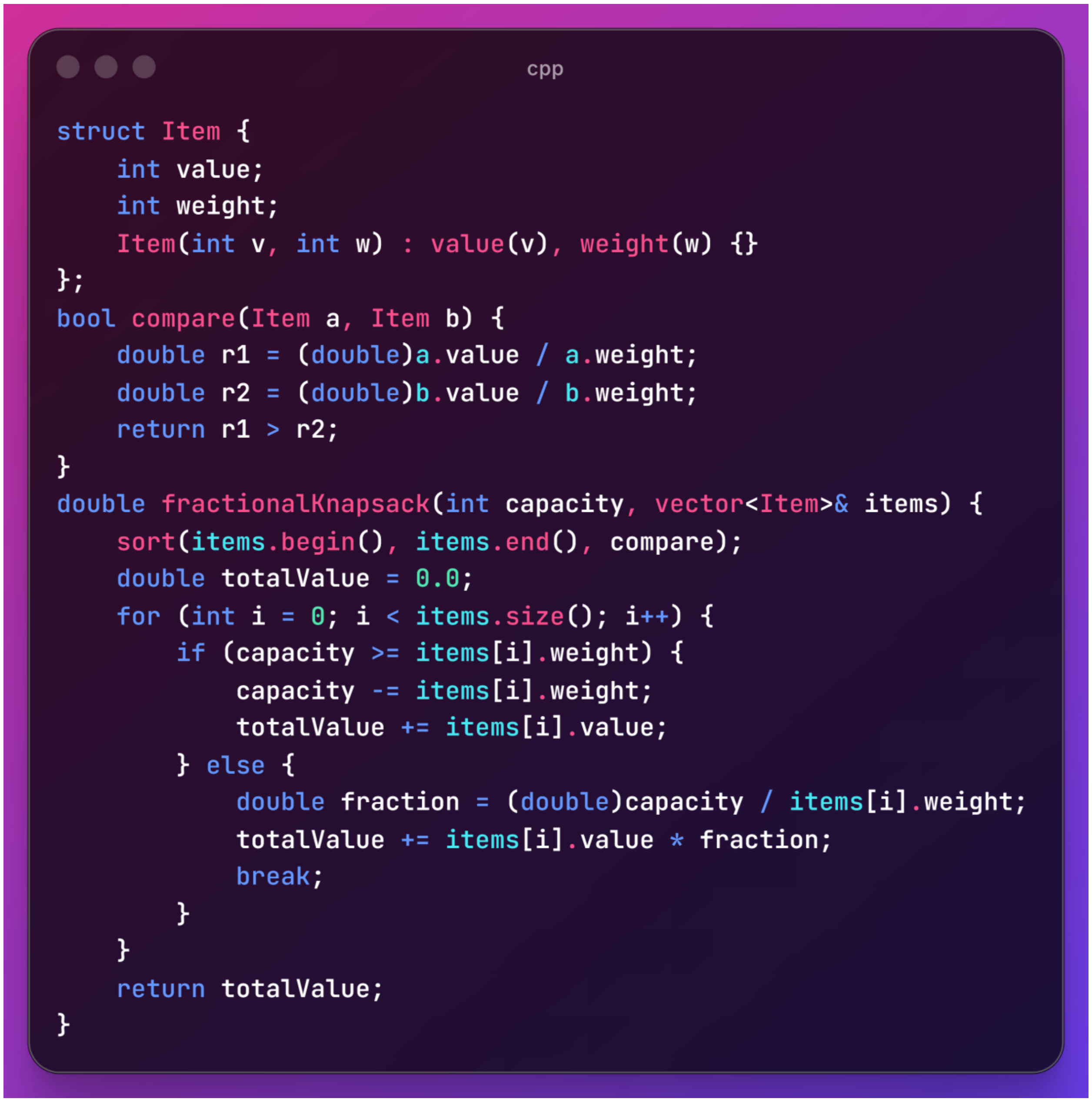


```cpp
int knapsack(int capacity, vector<int>& weights, vector<int>& values, int n) {
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
              dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];
            }
        }
    }
    return dp[n][capacity];
}
vector<int> getSelectedItems(int capacity, vector<int>& weights, vector<int>& values, int n) {
    vector<vector<int>> dp(n + 1, vector<int>(capacity + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
            } else {
                dp[i][w] = dp[i - 1][w];

        }
    }
    vector<int> selected;
    int w = capacity;

    for (int i = n; i > 0; i--) {
        if (dp[i][w] != dp[i - 1][w]) {
            selected.push_back(i - 1);
            w -= weights[i - 1];
        }
    }
    return selected;
}
```

Time & Space Complexity (0/1 Knapsack):

- Time : O(n * W)
- Space : O(n * W)

Code for implementation (Fractional Knapsack) :



cpp

```cpp
struct Item {
    int value;
    int weight;
    Item(int v, int w) : value(v), weight(w) {}
};
bool compare(Item a, Item b) {
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}
double fractionalKnapsack(int capacity, vector<Item>& items) {
    sort(items.begin(), items.end(), compare);
    double totalValue = 0.0;
    for (int i = 0; i < items.size(); i++) {
        if (capacity >= items[i].weight) {
            capacity -= items[i].weight;
            totalValue += items[i].value;
        } else {
            double fraction = (double)capacity / items[i].weight;
            totalValue += items[i].value * fraction;
            break;
        }
    }
    return totalValue;
}
```

Time & Space Complexity (Fractional Knapsack):

- Time : O(nlog n) for sorting
- Space : O(1) (if sorted in place)

Conclusion:

In this lab we were demonstratd the idea of dynamic programming through the 0/1 Knapsack problem. DP helps to solve those problems where normal greedy or recursion method fails. Because many subproblems are repeated here. We also studied the fractional knapsack problem along with the 0/1 knapsack , which is easier to solve and solved using greedy approach. By implementing these methods, we understood how DP helps to optimize decisions in many real-life problems.

Reference :

1. Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, MIT Press.
2. GeeksforGeeks: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
3. GeeksforGeeks: https://www.geeksforgeeks.org/fractional-knapsack-problem/
4. Programiz: https://www.programiz.com/dsa/dynamic-programming
5. Class Lecture

Submitted By,
Name : Saleh Sadid Mir
Roll : 2207024
Batch : 2k22
Group : A1
Year : Second
Term : Second

Date of performance : 12-11-25
Date of Submission : 05-12-25