PBLE Report

Semester	S.E. Semester III – Computer Engineering
Subject	Analysis of Algorithm
Subject Professor In-charge	Prof. Swapnil S. Sonawane

Roll Number	Name of Students
24102C0086	Jay Keluskar
24102C0075	Rishi Shah
25102C2008	Samarth Pagaria

Name of the Project:

Warehouse Packing Optimization – Maximize value of stored goods within weight/space limit. Use Cases: (a) Knapsack packing. (b) Compare 0/1 vs Fractional. (c) Add Branch & Bound optimization.

Project Description:

The *Warehouse Packing Optimization* project aims to determine the most efficient way to store items in a warehouse so that the total value of goods is maximized without exceeding the available weight or space capacity. Each item has two parameters — weight and value — and the system must decide how much of each item to include to achieve the highest possible total value.

This project applies the Fractional Knapsack Algorithm, a Greedy strategy that selects items based on their value-to-weight ratio. The algorithm prioritizes items that offer the highest value per unit of weight and allows taking fractions of items when the remaining capacity cannot accommodate the entire item. This approach ensures that the solution is optimal and computationally efficient, making it ideal for real-time warehouse or logistics operations.

Project Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// --- Constraints defined by user ---
#define MAX_ITEMS 15
#define MAX_CAPACITY 100
// --- Global Variable and Data Structures ---
// Global variable to track the best 0/1 solution found by B&B
int max_profit_bb = 0;
typedef struct {
    int id; // Original item ID for tracking
    int weight;
    int value;
    double ratio;
 Item;
typedef struct Node {
    int level;
    int profit;
    int weight;
    double bound;
 Node;
// --- 1. HELPER FUNCTIONS ---
// Comparison function for qsort: sorts items in DESCENDING order of ratio
int compare_items_ratio(const void *a, const void *b) {
    Item *itemA = (Item *)a;
    Item *itemB = (Item *)b;
    // Sorts high ratio first (A > B means A comes first)
    if (itemA->ratio < itemB->ratio) return 1;
    if (itemA->ratio > itemB->ratio) return -1;
    return 0;
// Calculates the Upper Bound for a node using Fractional Knapsack logic on
remaining items
double compute_bound(Node u, int n, int W, Item arr[]) {
    int j;
    int current_weight = u.weight;
    double result = u.profit;
    int remaining_capacity = W - current_weight;
    for (j = u.level + 1; j < n; j++) {
        if (remaining_capacity <= 0) break;
        if (arr[j].weight <= remaining_capacity) {
            remaining_capacity -= arr[j].weight;
            result += arr[j].value;
```

```
        else {
            // Take a fraction of the last item
            result += (double)remaining_capacity * arr[j].ratio;
            remaining_capacity = 0;
    return result;
// --- 2. 0/1 KNAPSACK (DYNAMIC PROGRAMMING) ---
// Addresses Use Case (a): Knapsack Packing
int knapsack01_dp(Item items[], int n, int capacity) {
    int i, w;
    // DP table size is (n+1) x (capacity+1)
    int dp[MAX_ITEMS + 1][MAX_CAPACITY + 1];
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            else if (items[i - 1].weight <= w) {
                int include = items[i - 1].value + dp[i - 1][w - items[i -
1].weight];
                int exclude = dp[i - 1][w];
                // --- Decision/Optimal Substructure (Conceptually "Merging") ---
                // Choose the maximum of including or excluding the current item
                dp[i][w] = (include > exclude) ? include : exclude;
            else {
                // --- Suboptimal Discard/Selection (Conceptually "Purging") ---
                // Item is too heavy, so we must exclude it. We only keep the
prior best value.
                dp[i][w] = dp[i - 1][w];
    return dp[n][capacity];
// --- 3. FRACTIONAL KNAPSACK (GREEDY) ---
// Addresses Use Case (b): Comparison with 0/1
double fractional_knapsack_greedy(Item items[], int n, int capacity) {
    int i;
    // Calculate ratio and sort for the Greedy selection
    for (i = 0; i < n; i++) {
        items[i].ratio = (double)items[i].value / items[i].weight;
    qsort(items, n, sizeof(Item), compare_items_ratio);
    double total_value = 0.0;
    int remaining_capacity = capacity;
    for (i = 0; i < n; i++) {
        if (remaining_capacity <= 0) break;
        if (items[i].weight <= remaining_capacity) {
            remaining_capacity -= items[i].weight;
            total_value += items[i].value;
        else {
            // Take the necessary fraction
            double fraction = (double)remaining_capacity / items[i].weight;
            total_value += items[i].value * fraction;
```

```
            remaining_capacity = 0;
    return total_value;
// --- 4. 0/1 KNAPSACK (RECURSIVE BRANCH & BOUND LOGIC) ---
// Addresses Use Case (c): Add Branch & Bound optimization
void knapsack01_bb_recursive_logic(Item items[], int n, int W, int level, int
current_weight, int current_profit) {
    if (level == n) {
        if (current_profit > max_profit_bb) {
            max_profit_bb = current_profit;
        return;
    // --- Branch 1: INCLUDE the current item ---
    int next_item_index = level;
    int weight_with_item = current_weight + items[next_item_index].weight;
    int profit_with_item = current_profit + items[next_item_index].value;
    if (weight_with_item <= W) {
        if (profit_with_item > max_profit_bb) {
            max_profit_bb = profit_with_item;
        // Check bound for the child node
        Node V_include;
        V_include.level = level;
        V_include.weight = weight_with_item;
        V_include.profit = profit_with_item;
        double bound = compute_bound(V_include, n, W, items);

        // --- Pruning: Cut the branch if the bound is not promising ---
        if (bound > max_profit_bb) {
            knapsack01_bb_recursive_logic(items, n, W, level + 1,
weight_with_item, profit_with_item);
    // --- Branch 2: EXCLUDE the current item ---
    // Check bound for the child node
    Node V_exclude;
    V_exclude.level = level;
    V_exclude.weight = current_weight;
    V_exclude.profit = current_profit;
    double bound = compute_bound(V_exclude, n, W, items);
    // --- Pruning: Cut the branch if the bound is not promising ---
    if (bound > max_profit_bb) {
        knapsack01_bb_recursive_logic(items, n, W, level + 1, current_weight,
current_profit);
// Main Wrapper Function for B&B
int knapsack01_bb(Item items[], int n, int W) {
    int i;
    // Reset the global max profit before starting the search
    max_profit_bb = 0;
    // Pre-processing: Calculate ratio and sort for effective bound calculation
    for (i = 0; i < n; i++) {
        items[i].ratio = (double)items[i].value / items[i].weight;
```

```
    qsort(items, n, sizeof(Item), compare_items_ratio);
    // Start the recursive search from the first item (level 0)
    knapsack01_bb_recursive_logic(items, n, W, 0, 0, 0);
    return max_profit_bb;
// --- MAIN PROGRAM AND COMPARISON ---
int main() {
    int n, capacity;
    int i;
    int value_dp, value_bb;
    double value_greedy;
    printf("Enter number of items (max %d): ", MAX_ITEMS);
    if (scanf("%d", &n) != 1 || n <= 0 || n > MAX_ITEMS) {
        printf("Invalid number of items. Exiting.\n");
        return 1;
    Item items_dp[MAX_ITEMS];
    Item items_greedy[MAX_ITEMS];
    Item items_bb[MAX_ITEMS];
    printf("Enter capacity of warehouse (max %d): ", MAX_CAPACITY);
    if (scanf("%d", &capacity) != 1 || capacity <= 0 || capacity > MAX_CAPACITY)
{
        printf("Invalid capacity. Exiting.\n");
        return 1;
    for (i = 0; i < n; i++) {
        printf("Enter weight and value of item %d: ", i + 1);
        if (scanf("%d %d", &items_dp[i].weight, &items_dp[i].value) != 2 ||
items_dp[i].weight <= 0 || items_dp[i].value < 0) {
            printf("Invalid input for item. Exiting.\n");
            return 1;
        // Copy the item data for all algorithms
        items_dp[i].id = i + 1;
        items_greedy[i] = items_bb[i] = items_dp[i];
    printf("\n--- Warehouse Packing Optimization: Max Value Comparison ---\n");
    // --- EXECUTE 0/1 KNAPSACK (DP) ---
    value_dp = knapsack01_dp(items_dp, n, capacity);
    // --- EXECUTE FRACTIONAL KNAPSACK (GREEDY) ---
    value_greedy = fractional_knapsack_greedy(items_greedy, n, capacity);
    // --- EXECUTE 0/1 KNAPSACK (BRANCH & BOUND) ---
    value_bb = knapsack01_bb(items_bb, n, capacity);
    // --- DISPLAY COMPARISON ---
    printf("| Method                  | Max Value |\n");
    printf("|-------------------------|-----------|\n");
    printf("| 0/1 Knapsack (DP)       | %-9d |\n", value_dp);
    printf("| Fractional Knapsack (G) | %-9.2f |\n", value_greedy);
    printf("| 0/1 Knapsack (B&B)      | %-9d |\n", value_bb);
    printf("|-------------------------|-----------|\n");
    return 0;
```

1. Summary Table — When Program Exits 🚫

Input Stage	Condition Causing Exit	Message
Number of items (n)	Non-integer, n <= 0, or n > 15	Invalid number of items. Exiting.
Item weight/value	Non-integer or missing values, or W <= 0, V < 0	Invalid input for item. Exiting.
Warehouse capacity	Non-integer, capacity <= 0, or capacity > 100	Invalid capacity. Exiting.

2. Summary Table — When Program Succeeds ✅

Input Stage	Condition for Success	Description / Outcome
Number of items (n)	Integer, 0 < n <= 15	Program accepts and processes the number of items.
Item weight/value	Integer values provided for all items, W > 0, V \ge 0	Program reads weights and values correctly.
Warehouse capacity	Integer, 0 < capacity <= 100	Program accepts capacity and runs all three algorithms (DP, Greedy, B&B).
Overall constraints	Feasible inputs provided (optional, algorithms handle exceeding weights)	Program calculates maximum value that can be stored without exceeding capacity using all three methods.
Computation	All three algorithms complete execution	Output displays the maximum value achievable for each method for comparison.

Output Screenshots:

1. **Testcase A :**

```
Enter number of items (max 15): 7
Enter capacity of warehouse (max 100): 100
Enter weight and value of item 1: 35 700
Enter weight and value of item 2: 25 500
Enter weight and value of item 3: 56 783
Enter weight and value of item 4: 81 900
Enter weight and value of item 5: 75 100
Enter weight and value of item 6: 54 280
Enter weight and value of item 7: 80 990

--- Warehouse Packing Optimization: Max Value Comparison ---
| Method                  | Max Achievable Value |
|-------------------------|----------------------|
| 0/1 Knapsack (DP)       | 1483                 |
| Fractional Knapsack (G) | 1759.29              |
| 0/1 Knapsack (B&B)      | 1483                 |
|-------------------------|----------------------|
```

2. **Testcase B :**

```
Enter number of items (max 15): 3
Enter capacity of warehouse (max 100): 221
Invalid capacity. Exiting.

Enter number of items (max 15): 20
Invalid number of items. Exiting.
```

GitHub Repository Link of Project: https://github.com/Jayz-yuors/AOA-Project.git