|  |  |
| --- | --- |
|  | **DEPARTMENT OF COMPUTER ENGINEERING** |

**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 |
| 24102A0024 | PRATIKSHA MUJMULE |
| 24102A0026 | AYYAN RAJAPKAR |
| 24102A0030 | ANAS SHAIKH |
|  |  |

**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:**

**Warehouse Packing Solution: Description**

**A warehouse packing solution is a systematic approach to efficiently organize and store items in a warehouse. Its primary goal is to maximize space utilization, reduce handling time, and increase profitability. The solution evaluates the size, weight, and quantity of items, and determines the optimal placement and sequence for packing.**

**Key Features:**

* **Optimized Storage: Items are arranged to use warehouse space efficiently.**
* **Profit Maximization: Packing decisions consider the value of items to achieve the highest profit.**
* **Inventory Management: Tracks items to prevent overstocking or shortages.**
* **Operational Efficiency: Reduces time for picking, packing, and shipping items.**
* **Decision Support: Provides recommendations for item positions and packing order.**

**Applications:**

* **E-commerce fulfillment centers**
* **Manufacturing and distribution warehouses**
* **Cold storage or perishable goods storage**

**In short, a warehouse packing solution ensures that items are stored safely, efficiently, and profitably, making warehouse operations smoother and more cost-effective.**

**Project Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**// ------------------- COMMON FUNCTIONS -------------------**

**// Max function**

**int max(int a, int b) {**

**return (a > b) ? a : b;**

**}**

**// Structure for items**

**struct Item {**

**int weight;**

**int value;**

**float ratio;**

**};**

**// ------------------- 0/1 KNAPSACK (DP) -------------------**

**int knapSack01(int W, int wt[], int val[], int n) {**

**int i, w;**

**int dp[30][30]; // Adjust size if needed**

**for (i = 0; i <= n; i++) {**

**for (w = 0; w <= W; w++) {**

**if (i == 0 || w == 0)**

**dp[i][w] = 0;**

**else if (wt[i - 1] <= w)**

**dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);**

**else**

**dp[i][w] = dp[i - 1][w];**

**}**

**}**

**printf("\n[0/1 Knapsack] Profit Table:\n");**

**for (i = 0; i <= n; i++) {**

**for (w = 0; w <= W; w++) {**

**printf("%3d ", dp[i][w]);**

**}**

**printf("\n");**

**}**

**printf("\nMaximum Profit (0/1): %d\n", dp[n][W]);**

**return dp[n][W];**

**}**

**// ------------------- FRACTIONAL KNAPSACK -------------------**

**void fractionalKnapsack(int W, int wt[], int val[], int n) {**

**int i, j;**

**float ratio[30], temp;**

**int tempInt;**

**float totalValue = 0.0;**

**// Calculate ratios**

**for (i = 0; i < n; i++) {**

**ratio[i] = (float)val[i] / wt[i];**

**}**

**// Sort by ratio**

**for (i = 0; i < n - 1; i++) {**

**for (j = i + 1; j < n; j++) {**

**if (ratio[i] < ratio[j]) {**

**temp = ratio[i]; ratio[i] = ratio[j]; ratio[j] = temp;**

**tempInt = val[i]; val[i] = val[j]; val[j] = tempInt;**

**tempInt = wt[i]; wt[i] = wt[j]; wt[j] = tempInt;**

**}**

**}**

**}**

**printf("\n[Fractional Knapsack] Items (sorted by ratio):\n");**

**for (i = 0; i < n; i++) {**

**printf("Item %d: value=%d, weight=%d, ratio=%.2f\n", i + 1, val[i], wt[i], ratio[i]);**

**}**

**// Compute maximum profit**

**for (i = 0; i < n; i++) {**

**if (wt[i] <= W) {**

**W -= wt[i];**

**totalValue += val[i];**

**} else {**

**totalValue += ratio[i] \* W;**

**break;**

**}**

**}**

**printf("\nMaximum Profit (Fractional): %.2f\n", totalValue);**

**}**

**// ------------------- BRANCH & BOUND KNAPSACK -------------------**

**struct Node {**

**int level;**

**int profit;**

**int weight;**

**float bound;**

**};**

**int compare(const void \*a, const void \*b) {**

**struct Item \*x = (struct Item \*)a;**

**struct Item \*y = (struct Item \*)b;**

**if (y->ratio > x->ratio) return 1;**

**else return -1;**

**}**

**// Bound function**

**float bound(struct Node u, int n, int W, struct Item arr[]) {**

**if (u.weight >= W) return 0;**

**float profit\_bound = (float)u.profit;**

**int j = u.level + 1;**

**int totweight = u.weight;**

**while ((j < n) && (totweight + arr[j].weight <= W)) {**

**totweight += arr[j].weight;**

**profit\_bound += arr[j].value;**

**j++;**

**}**

**if (j < n)**

**profit\_bound += (W - totweight) \* arr[j].ratio;**

**return profit\_bound;**

**}**

**int knapsackBB(int W, struct Item arr[], int n) {**

**qsort(arr, n, sizeof(struct Item), compare);**

**struct Node u, v;**

**u.level = -1;**

**u.profit = 0;**

**u.weight = 0;**

**float maxProfit = 0;**

**u.bound = bound(u, n, W, arr);**

**struct Node queue[100];**

**int front = 0, rear = 0;**

**queue[rear++] = u;**

**while (front < rear) {**

**u = queue[front++];**

**if (u.level == n - 1) continue;**

**v.level = u.level + 1;**

**// Case 1: include item**

**v.weight = u.weight + arr[v.level].weight;**

**v.profit = u.profit + arr[v.level].value;**

**if (v.weight <= W && v.profit > maxProfit)**

**maxProfit = v.profit;**

**v.bound = bound(v, n, W, arr);**

**if (v.bound > maxProfit)**

**queue[rear++] = v;**

**// Case 2: exclude item**

**v.weight = u.weight;**

**v.profit = u.profit;**

**v.bound = bound(v, n, W, arr);**

**if (v.bound > maxProfit)**

**queue[rear++] = v;**

**}**

**printf("\nMaximum Profit (Branch & Bound): %d\n", (int)maxProfit);**

**return (int)maxProfit;**

**}**

**// ------------------- MAIN -------------------**

**int main() {**

**int n, W, i, choice;**

**int wt[30], val[30];**

**struct Item items[30];**

**printf("Enter number of items: ");**

**scanf("%d", &n);**

**printf("Enter weights and values of items:\n");**

**for (i = 0; i < n; i++) {**

**printf("Item %d - Weight: ", i + 1);**

**scanf("%d", &wt[i]);**

**printf("Item %d - Value: ", i + 1);**

**scanf("%d", &val[i]);**

**items[i].weight = wt[i];**

**items[i].value = val[i];**

**items[i].ratio = (float)val[i] / wt[i];**

**}**

**printf("Enter maximum capacity of knapsack: ");**

**scanf("%d", &W);**

**printf("\nChoose Method:\n");**

**printf("1. 0/1 Knapsack (DP)\n");**

**printf("2. Fractional Knapsack (Greedy)\n");**

**printf("3. Branch and Bound Knapsack\n");**

**printf("Enter choice: ");**

**scanf("%d", &choice);**

**if (choice == 1)**

**knapSack01(W, wt, val, n);**

**else if (choice == 2)**

**fractionalKnapsack(W, wt, val, n);**

**else if (choice == 3)**

**knapsackBB(W, items, n);**

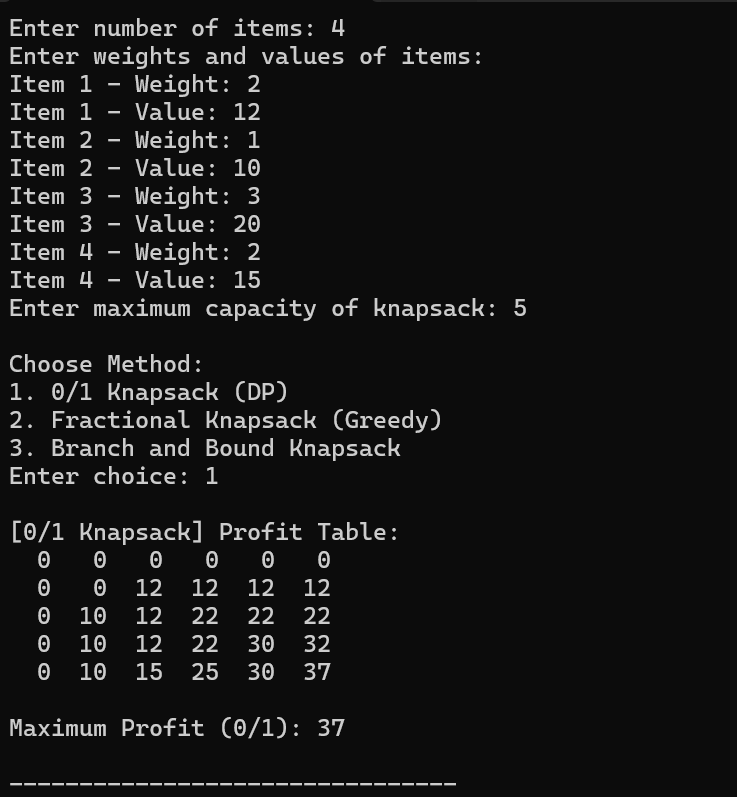
**else**

**printf("Invalid choice!\n");**

**return 0;**

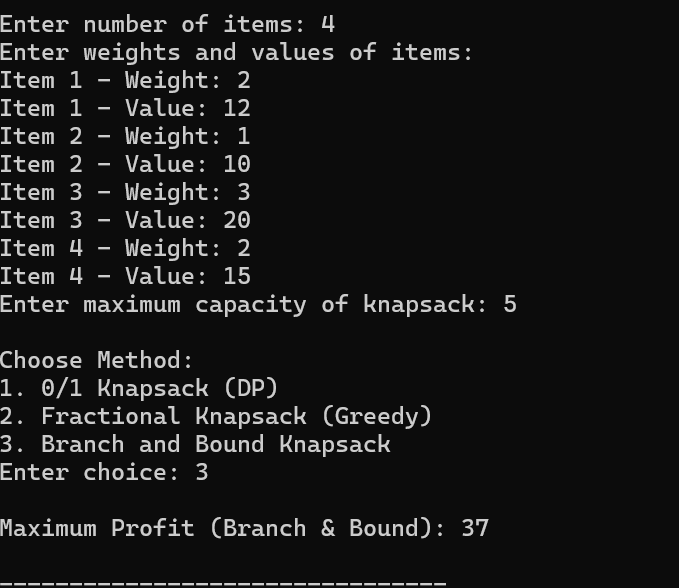
**}**

**Output Screenshots:**

****

**A screenshot of a computer program

AI-generated content may be incorrect.**

****

**GitHub Repository Link of Project:**