|  |  |  |  |
| --- | --- | --- | --- |
|  | Bansilal Ramnath Agarwal Charitable Trust's  Vishwakarma Institute of Information Technology  **Department of**  **Artificial Intelligence and Data Science** | | |
| Name: Siddhesh Dilip Khairnar | | | |
| Class: TY | Division: B | | Roll No: 372028 |
| Semester: V | | Academic Year: 2023-24 | |
| Subject Name & Code: Design and Analysis of Algorithm: ADUA31202 | | | |
| Title of Assignment: Implement 0/1 Knapsack problem using Following algorithmic strategies.   1. Dynamic programming 2. Back tracking 3. Branch and bound | | | |

**ASSIGNMENT NO. 5**

A piece of paper with writing on it

Description automatically generated

A piece of paper with writing on it

Description automatically generated

A piece of paper with writing on it

Description automatically generated

A piece of paper with writing on it

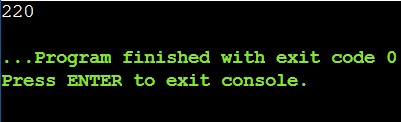
Description automatically generated

**Using Dynamic Programming approach**

**Program Code:**

|  |
| --- |
| #include <bits/stdc++.h> using namespace std;  int max(int a, int b)  { return (a > b) ? a : b;  } int knapSack(int W, int wt[], int val[], int n)  { int i, w; vector<vector<int>> K(n + 1, vector<int>(W + 1));  for(i = 0; i <= n; i++)  { for(w = 0; w <= W; w++)  {  if (i == 0 || w == 0) K[i][w] = 0;  else if (wt[i - 1] <= w)  K[i][w] = max(val[i - 1] +  K[i - 1][w - wt[i - 1]], K[i - 1][w]); else  K[i][w] = K[i - 1][w];  } }  return K[n][W];  }  int main()  { int val[] = { 60, 100, 120 }; int wt[] = { 10, 20, 30 }; int W = 50;  int n = sizeof(val) / sizeof(val[0]); cout << knapSack(W, wt, val, n); return 0;  } |

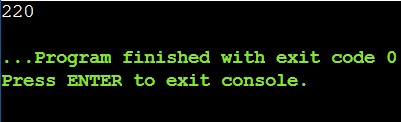
**Output:**



**Using Backtracking approach Program Code:**

|  |
| --- |
| #include <bits/stdc++.h> using namespace std; int knapSack(int W, int wt[], int val[], int n)  { int dp[W + 1]; memset(dp, 0, sizeof(dp));  for (int i = 1; i < n + 1; i++) { for (int w = W; w >= 0; w--) {  if (wt[i - 1] <= w) dp[w] = max(dp[w], dp[w - wt[i - 1]] + val[i - 1]);  } }  return dp[W];  }  int main()  {  int val[] = { 60, 100, 120 }; int wt[] = { 10, 20, 30 }; int W = 50; int n = sizeof(val) / sizeof(val[0]); cout << knapSack(W, wt, val, n); return 0;  } |

**Output:**



**Using Branch and Bound approach**

This algorithm uses the greedy approach to calculate an upper bound on the solution. If a node's upper bound is less than the maxProfit, we don't explore it further because it cannot lead to a better solution. This helps us prune the search space and find the optimal solution more efficiently.

In essence, the algorithm explores different combinations of items, always considering the most promising ones first, and updates the maxProfit when a better solution is found. This way, it finds the best solution for the Fractional Knapsack problem.

**Algorithm:**

1. Sort all items in decreasing order of the value-to-weight ratio. This allows us to consider the most valuable items first.
2. Initialize maxProfit to 0. This will keep track of the best solution found so far.
3. Create an empty queue, Q, to explore different possibilities.
4. Start with a dummy node representing the root of a decision tree. This node has zero profit and zero weight.
5. While there are nodes in the queue Q:
   * Take out an item from Q. Let's call it u.
   * Calculate the profit of the next level node. If this profit is greater than maxProfit, update maxProfit.
   * Calculate a bound for the next level node. If this bound is greater than maxProfit, add the next level node to the queue.
   * Consider the scenario where the next level node is not included in the solution and add a new node to the queue with an increased level, but without the weight and profit of the next level nodes.

**Program Code:**

|  |
| --- |
| #include <bits/stdc++.h> using namespace std;  struct Item  { float weight; int value;  }; struct Node  { int level, profit, bound; float weight;  }; bool cmp(Item a, Item b)  { double r1 = (double)a.value / a.weight; double r2 = (double)b.value / b.weight; return r1 > r2;  } int bound(Node u, int n, int W, Item arr[])  { if (u.weight >= W) return 0;  int profit\_bound = 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].value / arr[j].weight;  return profit\_bound;  } |

|  |
| --- |
| int knapsack(int W, Item arr[], int n)  { sort(arr, arr + n, cmp);    queue<Node> Q;  Node u, v;    u.level = -1;  u.profit = u.weight = 0;  Q.push(u); int maxProfit = 0; while (!Q.empty())  {  u = Q.front();  Q.pop(); if (u.level == -1)  v.level = 0; if (u.level == n-1) continue;    v.level = u.level + 1;    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)  Q.push(v);    v.weight = u.weight;  v.profit = u.profit;  v.bound = bound(v, n, W, arr); if (v.bound > maxProfit)  Q.push(v);  } return maxProfit;  } int main() |
| {  int W = 10;  Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},  {5, 95}, {3, 30}}; int n = sizeof(arr) / sizeof(arr[0]);  cout << "Maximum possible profit = "  << knapsack(W, arr, n);  return 0;  } |

**Output:**

