|  |  |
| --- | --- |
| **Paths of 0-1 knapsack In C++** | |
| #include <iostream>  #include <vector>  #include <deque>  using namespace std;  struct Pair {      int i;      int j;      string psf;      Pair(int i, int j, string psf) {          this->i = i;          this->j = j;          this->psf = psf;      }  };  void printPaths(vector<vector<int>>& dp, vector<int>& vals, vector<int>& wts, int i, int j, string psf, deque<Pair>& que) {      while (!que.empty()) {          Pair rem = que.front();          que.pop\_front();          if (rem.i == 0 || rem.j == 0) {              cout << rem.psf << endl;          } else {              int exc = dp[rem.i - 1][rem.j];              if (rem.j >= wts[rem.i - 1]) {                  int inc = dp[rem.i - 1][rem.j - wts[rem.i - 1]] + vals[rem.i - 1];                  if (dp[rem.i][rem.j] == inc) {                      que.push\_back(Pair(rem.i - 1, rem.j - wts[rem.i - 1], to\_string(rem.i - 1) + " " + rem.psf));                  }              }              if (dp[rem.i][rem.j] == exc) {                  que.push\_back(Pair(rem.i - 1, rem.j, rem.psf));              }          }      }  }  void knapsackPaths(vector<int>& vals, vector<int>& wts, int cap) {      int n = vals.size();      vector<vector<int>> dp(n + 1, vector<int>(cap + 1, 0));      for (int i = 1; i <= n; i++) {          for (int j = 1; j <= cap; j++) {              dp[i][j] = dp[i - 1][j];              if (j >= wts[i - 1]) {                  dp[i][j] = max(dp[i][j], dp[i - 1][j - wts[i - 1]] + vals[i - 1]);              }          }      }      int ans = dp[n][cap];      cout << "Maximum value: " << ans << endl;      deque<Pair> que;      que.push\_back(Pair(n, cap, ""));      printPaths(dp, vals, wts, n, cap, "", que);  }  int main() {      vector<int> vals = {15, 14, 10, 45, 30};      vector<int> wts = {2, 5, 1, 3, 4};      int cap = 7;      knapsackPaths(vals, wts, cap);      return 0;  } | **Dry Run Using a Table******Step 1: Initialize DP Table**** We define a **DP table (dp[i][j])**, where:   * dp[i][j] = **Maximum value** that can be obtained using the first i items with a capacity j.  ****Step 1.1: Base Case****  * If i = 0 (no items), or j = 0 (zero capacity), dp[i][j] = 0.  ****Step 1.2: Fill the DP Table**** If including the item **does not exceed capacity**, we check:   * Exclude item i → dp[i-1][j] * Include item i → dp[i-1][j - wts[i-1]] + vals[i-1]  | **i\j** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | **1** (val=15, wt=2) | 0 | 0 | 15 | 15 | 15 | 15 | 15 | 15 | | **2** (val=14, wt=5) | 0 | 0 | 15 | 15 | 15 | 15 | 15 | 15 | | **3** (val=10, wt=1) | 0 | 10 | 15 | 25 | 25 | 25 | 25 | 25 | | **4** (val=45, wt=3) | 0 | 10 | 15 | 45 | 55 | 60 | 70 | 70 | | **5** (val=30, wt=4) | 0 | 10 | 15 | 45 | 55 | 60 | 70 | **75** |   The **maximum value** obtained is 75 at dp[5][7]. ****Step 2: Print All Paths**** Using **backtracking**, the function printPaths reconstructs paths that lead to dp[n][cap] = 75. ****Backtracking Paths****  1. Start at dp[5][7] = 75    * dp[4][3] = 45 → Item 5 (index 4, value 30, weight 4) is included. 2. Now at dp[4][3] = 45    * dp[3][0] = 0 → Item 4 (index 3, value 45, weight 3) is included.   Thus, one of the optimal selections is {30, 45}. ****Final Output**** Maximum value: 75  4 3 |
| Output:- Maximum value: 75  3 4 | |