class Knapsack {

static int solveKnapsack(int[] profits, int[] weights, int capacity) {

//TODO: Write - Your - Code

return -1;

}

}

SOLUTION:

class Knapsack {

static int solveKnapsack(int[] profits, int[] weights, int capacity) {

// basic checks

if (capacity <= 0 || profits.length == 0 || weights.length != profits.length)

return 0;

int n = profits.length;

// we only need one previous row to find the optimal solution, overall we need '2' rows

// the above solution is similar to the previous solution, the only difference is that

// we use `i%2` instead if `i` and `(i-1)%2` instead if `i-1`

int[][] dp = new int[2][capacity+1];

// if we have only one weight, we will take it if it is not more than the capacity

for(int c=0; c <= capacity; c++) {

if(weights[0] <= c)

dp[0][c] = dp[1][c] = profits[0];

}

// process all sub-arrays for all the capacities

for(int i=1; i < n; i++) {

for(int c=0; c <= capacity; c++) {

int profit1= 0, profit2 = 0;

// include the item, if it is not more than the capacity

if(weights[i] <= c)

profit1 = profits[i] + dp[(i-1)%2][c-weights[i]];

// exclude the item

profit2 = dp[(i-1)%2][c];

// take maximum

dp[i%2][c] = Math.max(profit1, profit2);

}

}

return dp[(n-1)%2][capacity];

}

}