**import heapq**

**# Class to store item information (weight, value, and index)**

**class Item:**

**def \_\_init\_\_(self, weight, value, index):**

**self.weight = weight**

**self.value = value**

**self.index = index**

**# Value-to-weight ratio**

**self.ratio = value / weight**

**# Function to solve the knapsack problem using Branch and Bound**

**def knapsack\_branch\_and\_bound(weights, values, capacity):**

**n = len(weights)**

**# Create a list of items from the weights and values**

**items = [Item(weights[i], values[i], i) for i in range(n)]**

**# Sort items by value-to-weight ratio in descending order**

**items.sort(key=lambda x: x.ratio, reverse=True)**

**# Class to represent a node in the search tree**

**class Node:**

**def \_\_init\_\_(self, level, value, weight, bound):**

**self.level = level  # Index of the item**

**self.value = value  # Total value so far**

**self.weight = weight  # Total weight so far**

**self.bound = bound  # Upper bound on value of the node**

**# Compare nodes by bound (used for priority queue)**

**def \_\_lt\_\_(self, other):**

**return self.bound > other.bound**

**# Function to calculate the upper bound on the maximum possible profit**

**def bound(node):**

**# If the weight exceeds the capacity, bound is 0**

**if node.weight >= capacity:**

**return 0**

**# Start with current value**

**profit\_bound = node.value**

**j = node.level + 1**

**tot\_weight = node.weight**

**# Add items while the total weight does not exceed the capacity**

**while j < n and tot\_weight + items[j].weight <= capacity:**

**tot\_weight += items[j].weight**

**profit\_bound += items[j].value**

**j += 1**

**# Add fractional part of the next item, if there's space**

**if j < n:**

**profit\_bound += (capacity - tot\_weight) \* items[j].ratio**

**return profit\_bound**

**# Priority queue (max-heap) to explore the tree**

**priority\_queue = []**

**# Start with a dummy node (root)**

**v = Node(-1, 0, 0, 0)**

**v.bound = bound(v)**

**max\_profit = 0  # Variable to keep track of the maximum profit**

**# Add the dummy node to the priority queue**

**heapq.heappush(priority\_queue, v)**

**# Explore the priority queue**

**while priority\_queue:**

**# Pop the node with the highest bound**

**v = heapq.heappop(priority\_queue)**

**# If bound is higher than the current max profit and not the last item**

**if v.bound > max\_profit and v.level < n - 1:**

**# Create a node representing taking the next item**

**u = Node(v.level + 1,**

**v.value + items[v.level + 1].value,**

**v.weight + items[v.level + 1].weight,**

**0)**

**# If its weight is within capacity and the value is greater than the max profit, update max profit**

**if u.weight <= capacity and u.value > max\_profit:**

**max\_profit = u.value**

**# Calculate the bound for this node**

**u.bound = bound(u)**

**# If the bound is greater than the current max profit, add it to the priority queue**

**if u.bound > max\_profit:**

**heapq.heappush(priority\_queue, u)**

**# Create a node representing not taking the next item**

**u = Node(v.level + 1, v.value, v.weight, 0)**

**u.bound = bound(u)**

**# If its bound is better than the max profit, add it to the queue**

**if u.bound > max\_profit:**

**heapq.heappush(priority\_queue, u)**

**# Return the maximum profit found**

**return max\_profit**

**# Example usage**

**weights = [1, 2, 3, 5]**

**values = [1, 6, 10, 16]**

**capacity = 7**

**print(f"Maximum value using Branch and Bound: {knapsack\_branch\_and\_bound(weights, values, capacity)}")**