In [1]:

```python
#DAA PR : 04
#0/1 Knapsack problem using branch and bound strategy
def knapsack_dp(values, weights, capacity):
    n = len(values)

    # Create a 2D array to store the maximum value for each subproblem
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    # Build the dp array
    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                # Include the item and check if it's more beneficial
                dp[i][w] = max(dp[i - 1][w],
                               values[i - 1] + dp[i - 1][w - weights[i - 1]])
            else:
                # Exclude the item
                dp[i][w] = dp[i - 1][w]

    # The maximum value for the entire capacity will be
    #stored in dp[n][capacity]
    return dp[n][capacity]


# Example usage
if __name__ == "__main__":
    values = [60, 100, 120]  # Values of the items
    weights = [10, 20, 30]  # Corresponding weights of the items
    capacity = 50  # Knapsack capacity

    max_value = knapsack_dp(values, weights, capacity)
    print(f"Maximum value in the knapsack: {max_value}")

# Branch and Bound Strategy:
import heapq

class Node:
    def __init__(self, level, value, weight, bound):
        self.level = level
        self.value = value
        self.weight = weight
        self.bound = bound

    def __lt__(self, other):
        return self.bound > other.bound  # Max-heap

# Function to calculate the upper bound
def bound(u, n, capacity, values, weights):
    if u.weight >= capacity:
        return 0

    bound_value = u.value
    j = u.level + 1
    total_weight = u.weight

    while j < n and total_weight + weights[j] <= capacity:
        total_weight += weights[j]
        bound_value += values[j]
        j += 1

    if j < n:
```

```
        bound_value += (capacity - total_weight) *
        values[j] / weights[j]

    return bound_value

def knapsack_bb(values, weights, capacity):
    n = len(values)

    # Sort by value-to-weight ratio in decreasing order
    items = sorted(range(n), key=lambda i: values[i] / weights[i],
                   reverse=True)
    values = [values[i] for i in items]
    weights = [weights[i] for i in items]

    max_value = 0  # Max value we can carry
    pq = []
    u = Node(-1, 0, 0, 0)  # Starting node
    u.bound = bound(u, n, capacity, values, weights)
    heapq.heappush(pq, u)

    while pq:
        u = heapq.heappop(pq)

        if u.level == n - 1 or u.bound <= max_value:
            continue

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

        # Consider including the current item
        v.weight = u.weight + weights[v.level]
        v.value = u.value + values[v.level]

        if v.weight <= capacity and v.value > max_value:
            max_value = v.value

        v.bound = bound(v, n, capacity, values, weights)

        if v.bound > max_value:
            heapq.heappush(pq, v)

        # Consider excluding the current item
        v = Node(u.level + 1, u.value, u.weight, 0)
        v.bound = bound(v, n, capacity, values, weights)

        if v.bound > max_value:
            heapq.heappush(pq, v)

    return max_value


# Example usage
if __name__ == "__main__":
    values = [60, 100, 120]  # Values of the items
    weights = [10, 20, 30]  # Corresponding weights of the items
    capacity = 50  # Knapsack capacity

    max_value = knapsack_bb(values, weights, capacity)
    print(f"Maximum value in the knapsack: {max_value}")
```

```
Maximum value in the knapsack: 220
Maximum value in the knapsack: 220
```

In [ ]: