From the textbook, the subproblem:

LET $P_j(y)$ **DENOTE** the optimal solution to KNAP(j,y). Where KNAP(j,y) represents the problem maximize $\sum_{1 \le i \le j} p_i x_i$ subject to the constraint $\sum_{1 \le i \le j} w_i x_i \le y$ with the requirement that $x_i \in \{0, 1\}, 1 \le i \le j$.

The recursive formulation will be:
$P_j(y) = max\{P_{j-1}(y), P_{j-1}(y - w_j) + p_j\}$

Base Case:
$P_0(y) = 0$ for all $y \ge 0$
$P_i(y) = -\infty$ if y < 0

Pseudo-Code:

```
def knapsack_top_down(weights, values, capacity):
    n = len(weights)
    cache = {}

    def P(j, y):
        if j == 0 or y == 0:
            return 0

        if (j, y) not in cache:
            if weights[j - 1] > y:
                result = P(j - 1, y)
            else:
                exclude_item = P(j - 1, y)
                include_item = P(j - 1, y - weights[j - 1]) + values[j -
1]

                result = max(exclude_item, include_item)

        cache[(j, y)] = result
        return result

    optimal_value = P(n, capacity)
    return optimal_value
```

```python
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5

optimal_value = knapsack_top_down(weights, values, capacity)
print("Optimal Value (Top-Down):", optimal_value)
```

```python
def knapsack_bottom_up(weights, values, capacity):
    n = len(weights)
    cache = [[0] * (capacity + 1) for _ in range(n + 1)]

    for j in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[j - 1] > w:
                cache[j][w] = cache[j - 1][w]
            else:
                cache[j][w] = max(cache[j - 1][w], cache[j - 1][w -
weights[j - 1]] + values[j - 1])

    return cache[n][capacity]

weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 5

optimal_value_bottom_up = knapsack_bottom_up(weights, values, capacity)
print("Optimal Value (Bottom-Up):", optimal_value_bottom_up)
```