DSC 190 - Discussion 07

Problem 1.

Let's consider the following scenario: A thief has set their sights on robbing a house. They come prepared with a bag that has a limited capacity, denoted by W. Inside the house, there are n items, each with its own value and weight, represented in the *values* and *weights* arrays, respectively. We've previously discussed a similar problem in one of our earlier discussion sessions. However, there's a small twist to this problem compared to what we discussed previously: the thief faces a binary decision for each item. They can either choose to include an item in their bag, or they can opt to exclude it entirely. There's no middle ground where they can take a fraction of an item. The thief's ultimate goal is to maximize the total value of the stolen items. This classic problem is commonly known as the "0/1 knapsack problem".

a) Discuss why a greedy approach will not give an optimal solution for this problem. Give an example where the greedy approach would fail.

Solution: The greedy approach will not work for this problem because it does not consider the possibility of excluding items to achieve a globally optimal solution. Here's an example - Let us consider a knapsack with a total capacity $W = 40$.

- Item A: value = 60, weight = 1
- Item B: value = 100, weight = 2
- Item C: value = 120, weight = 3.

If we were to pick items greedily, we would start with the item that has the highest value-to-weight ratio. This would result in us picking item A followed by item B. This gives a total weight of 3 with an overall value of 160. We cannot pick item C further as the capacity of the bag would exceed. However, the optimal solution for this problem is to pick items A and C which would give an overall value of 180. The greedy solution has failed in this scenario.

b) Write a recursive function that takes in $values, weights, index$, and W and computes the maximum value of the items that the thief can steal.

Solution:

```python
def knapsack_recursive(weights, values, index, W):
    """
    Calculate the maximum value that can be obtained in
    the 0/1 knapsack problem using a recursive approach.

    :param weights: List of item weights.
    :param values: List of item values.
    :param index: Current index of the item being considered.
    :param W: Remaining weight capacity of the knapsack.
    :return: The maximum value that can be obtained.

    This function recursively explores two options for each item: either the item
    is picked (if its weight allows) or it's not picked.  It returns the maximum
    value that can be obtained with the given items and weight capacity.
    """
    # Base case: If we're at the first item, check if it can be added to the
    # knapsack.
    if index == 0:
        if weights[0] <= W:
            return values[0]
        else:
            return 0

    # Calculate the maximum value if the current item is not picked.
    not_picked = knapsack_recursive(weights, values, index - 1, W)

    # Initialize the maximum value if the current item is picked.
    picked = float('-inf')

    # Check if the current item can be added to the knapsack and calculate
    # the maximum value if picked.
    if weights[index] <= W:
        picked = values[index] + knapsack_recursive(weights, values,
                index - 1, W - weights[index])

    # Return the maximum value between the two options: picked or not picked.
    return max(picked, not_picked)

if __name__ == "__main__":
    weights = [1, 2, 3]
    values = [60, 100, 120]
    max_value = knapsack_recursive(weights, values, 2, 4)
    print("Maximum value:", max_value)
```

c) Optimize the above recursive solution using top-down dynamic programming.

Solution:

```
def top_down_knapsack(weights, values, index, W, n, cache=None):
    """
    Calculate the maximum value that can be obtained in the
    0/1 knapsack problem using a top-down dynamic programming approach with
    memoization.

    :param weights: List of item weights.
    :param values: List of item values.
    :param index: Current index of the item being considered.
    :param W: Remaining weight capacity of the knapsack.
    :param n: Total number of items.
    :param cache: Memoization cache to store previously computed
    results (optional).
    :return: The maximum value that can be obtained.

    This function calculates the maximum value that can be obtained
    in the 0/1 knapsack problem using a top-down dynamic programming approach
    with memoization. It considers two options for each item: either the item
    is picked (if its weight allows) or it's not picked. It returns the
    maximum value that can be obtained with the given items and
    weight capacity.
    """

    # Initialize the memoization cache if it's not provided.
    if cache is None:
        cache = [[None for j in range(W + 1)] for i in range(n)]

    # Base case: If we're at the first item, check if it can be added
    # to the knapsack.
    if index == 0:
        if weights[0] <= W:
            return values[0]
        else:
            return 0

    # Check if the result for the current index and remaining capacity
    # is already memoized.
    if cache[index][W] is not None:
        return cache[index][W]

    # Calculate the maximum value if the current item is not picked.
    not_picked = top_down_knapsack(weights, values, index - 1, W, n, cache)

    # Initialize the maximum value if the current item is picked.
    picked = float('-inf')

    # Check if the current item can be added to the knapsack and
    # calculate the maximum value if picked.
    if weights[index] <= W:
        picked = values[index] + top_down_knapsack(weights, values, index - 1,
                W - weights[index], n, cache)

    # Store the result in the memoization cache and return the maximum value.
    cache[index][W] = max(picked, not_picked)
    return cache[index][W]

if __name__ == "__main__":
    weights = [1, 2, 3]
    values = [60, 100, 120]
    max_value = top_down_knapsack(weights, values, 2, 4, len(weights))
    print("Maximum value:", max_value)
```

d) Write a bottom-up dynamic programming solution for the 0/1 knapsack problem.

Solution:

```
def bottom_up_knapsack(weights, values, index, W, n):
    """
    Calculate the maximum value that can be obtained in
    the 0/1 knapsack problem using a bottom-up dynamic programming
    approach.

    :param weights: List of item weights.
    :param values: List of item values.
    :param index: Current index of the item being considered.
    :param W: Remaining weight capacity of the knapsack.
    :param n: Total number of items.
    :return: The maximum value that can be obtained.
    This function calculates the maximum value that can be obtained
    in the 0/1 knapsack problem using a bottom-up dynamic programming approach.
    It fills in a cache (2D array) to store and compute results for different
    combinations of items and weight capacities.
    The final result is stored in the bottom-right corner of the cache.
    """

    # Create a cache (2D array) to store the maximum values for
    # different combinations of items and weight capacities.
    cache = [[0 for i in range(W + 1)] for j in range(n + 1)]

    # Iterate through each item and each possible weight capacity.
    for item in range(1, n + 1):
        for capacity in range(1, W + 1):
            if weights[item - 1] <= capacity:
                # If the item can be added to the knapsack, ↵
                # calculate the maximum value.
                cache[item][capacity] = max(
                    values[item - 1] + cache[item - 1][capacity - weights[item - 1]],
                    cache[item - 1][capacity]
                )
            else:
                # If the item is too heavy, just copy the value from the
                # previous row.
                cache[item][capacity] = cache[item - 1][capacity]

    # The result is stored in the bottom-right corner of the cache.
    return cache[n][W]
```