LAB REPORT

Assignment 3

NAME : P.SATHWIKA

REGISTRATION NUMBER : 21BCE8118

SUBJECT : Design and Analysis of Algorithms.

LAB SLOT : L53+L5

IMPLEMENT KNAPSACK PROBLEM

BRUTE FORCE APPROACH

The brute force approach to solving the Knapsack Problem involves generating all possible combinations of items and then selecting the combination that maximizes the value while not exceeding the weight capacity of the knapsack.

PSEUDOCODE :

Step1: Check if the current weight exceeds the capacity

if curr_weight > capacity:

return

Step 2: Check if all items are considered

if index == n:

Step 3: Update max_value if the current combination has a higher value

max_value = max(max_value, curr_val)

return

Step 4: Include the current item and move to the next item

generate_combinations(curr_val + values[index], curr_weight + weights[index], index + 1)

Step 5: Exclude the current item and move to the next item

generate_combinations(curr_val, curr_weight, index + 1)

Step 6: Call the recursive function with initial values

generate_combinations(0, 0, 0)

ALGORITHM :

Generate all combinations: Enumerate all possible subsets of items. This can be done using recursive backtracking or by using binary representations of numbers (e.g., bit manipulation).

Evaluate each combination: For each subset of items generated in step 1, calculate the total value and total weight of the items in the subset.

Check feasibility: Ensure that the total weight of each subset does not exceed the capacity of the knapsack. If the weight exceeds the capacity, discard that subset.

Select the best combination: Among the feasible combinations, select the one with the maximum total value.

Return the solution: Once all combinations have been evaluated, return the combination with the maximum total value as the solution to the Knapsack Problem.

CODE:

```
print("21bce8118")
def knapsack_bruteforce(weights, values, capacity):
    num_items = len(weights)
    max_value = 0
    best_combination = []

    # Generate all possible combinations using binary representation
    for i in range(2**num_items):
        current_combination = [int(bit) for bit in bin(i)[2:].zfill(num_items)]
        current_value = sum([current_combination[j] * values[j] for j in range(num_items)])
        current_weight = sum([current_combination[j] * weights[j] for j in range(num_items)])

        # Check if the combination is feasible and has a higher value
        if current_weight <= capacity and current_value > max_value:
            max_value = current_value
            best_combination = current_combination

    return best_combination, max_value

# Example usage
weights = [4, 3, 4, 5]
values = [42, 12, 40, 25]
capacity = 10

best_combination, max_value = knapsack_bruteforce(weights, values, capacity)
print("Best combination:", best_combination)
print("Maximum value:", max_value)
```

```python
print("21bce8118")
def knapsack_bruteforce(weights, values, capacity):
    num_items = len(weights)
    max_value = 0
    best_combination = []

    # Generate all possible combinations using binary representation
    for i in range(2**num_items):
        current_combination = [int(bit) for bit in bin(i)[2:].zfill(num_items)]
        current_value = sum([current_combination[j] * values[j] for j in range(num_items)])
        current_weight = sum([current_combination[j] * weights[j] for j in range(num_items)])

        # Check if the combination is feasible and has a higher value
        if current_weight <= capacity and current_value > max_value:
            max_value = current_value
            best_combination = current_combination

    return best_combination, max_value

# Example usage
weights = [4, 3, 4, 5]
values = [42, 12, 40, 25]
capacity = 10

best_combination, max_value = knapsack_bruteforce(weights, values, capacity)
print("Maximum value:", max_value)
```

OUTPUT:

```
21bce8118
Maximum value: 82
```