Assignment – 3

Ayyappa Reddy Maramreddy (R11888944)

1. **Implementation (30 points):** Implement both of these algorithms in a language of your choice. Make sure both of your algorithms work for any problem instance. Verify that your algorithms both have worst-case complexity of Θ(nW) as analysed in class.

   Deliverable 1: Your implementation's code and your verification of correctness.

Solution:

The Dynamic Programming can be done by using either bottom up approach or Top down approach.

Bottom-Up Approach:

In bottom-up approach, we start from the bottom and iteratively solve each of the subproblem and go our way to the top. Similar to the top down, we store the values of the already calculated subproblems solutions and store it in two-dimensional array.

```python
#1st implementing both the algorithms and checking their completeness
#First below is the bottom up approach here for Knapsack Problem
def knapsack_bottom_up(items_weight, items_value, knapsack_capacity):
    num_items = len(items_weight)
    dp_table = [[0 for _ in range(knapsack_capacity + 1)] for _ in range(num_items + 1)]

    for i in range(1, num_items + 1):
        for w in range(1, knapsack_capacity + 1):
            if items_weight[i - 1] <= w:
                dp_table[i][w] = max(items_value[i - 1] + dp_table[i - 1][w - items_weight[i - 1]], dp_table[i - 1][w])
            else:
                dp_table[i][w] = dp_table[i - 1][w]

    return dp_table[num_items][knapsack_capacity]
```

Fig1. Knapsack_bottom_up approach

Top-Down Approach: In top-down approach, we start with the problem and break into number of subproblems. We solve each of the sub problem recursively and then use a memoisation table to store the solutions of each subproblem. This is to avoid unnecessary calculations of subproblems which are already calculated. We can use the table to retrieve the already calculated subproblems instead of again calculating the solution.

```python
#Second below is the Top down approach here for Knapsack Problem
def knapsack_Top_down(items_weight, items_value, knapsack_capacity, current_index, memo):
    if current_index < 0 or current_index >= len(items_weight):
        return 0

    if (current_index, knapsack_capacity) in memo:
        return memo[(current_index, knapsack_capacity)]

    values = 0
    if items_weight[current_index] <= knapsack_capacity:
        values = items_value[current_index] + knapsack_Top_down(
            items_weight, items_value, knapsack_capacity - items_weight[current_index], current_index + 1, memo)

    values1 = knapsack_Top_down(items_weight, items_value, knapsack_capacity, current_index + 1, memo)

    memo[(current_index, knapsack_capacity)] = max(values, values1)
    return memo[(current_index, knapsack_capacity)]
```

Fig2. Knapsack_Top_down approach

```python
def knapsack_exhaustive_search(weights, values, capacity):
    n = len(weights)
    best_value = 0
    best_combination = None

    for combination in itertools.product([0, 1], repeat=n):
        total_weight = sum(combination[i] * weights[i] for i in range(n))
        total_value = sum(combination[i] * values[i] for i in range(n))

        if total_weight <= capacity and total_value > best_value:
            best_value = total_value
            best_combination = combination

    return best_value, best_combination
```

Fig3. Exhaustive Search Algorithm

```python
def main():
    items_weight = [4, 2, 6, 8]
    items_value = [15, 25, 40, 35]
    knapsack_capacity = 7

    # Verify correctness
    Bottomup_result = knapsack_bottom_up(items_weight, items_value, knapsack_capacity)
    Topdown_result = knapsack_Top_down(items_weight, items_value, knapsack_capacity, 0, {})

    print("Topdown approach Result:", Topdown_result)
    print("Bottomup approach Result:", Bottomup_result)

    if Bottomup_result == Topdown_result:
        print("Results match.")
    else:
        print("Results do not match.")

    # Test Exhaustive Search
    result_value, result_combination = knapsack_exhaustive_search(items_weight, items_value, knapsack_capacity)
    print("Best value (Exhaustive Search):", result_value)
    print("Selected items (Exhaustive Search):", result_combination)

if __name__ == "__main__":
    main()
```

Fig4. Verifying the correctness of both the algorithms

```
Topdown approach Result: 40
Bottomup approach Result: 40
Results match.
Best value (Exhaustive Search): 40
Selected items (Exhaustive Search): (0, 0, 1, 0)
```

Fig5. Result

Result: Here is the result for the above verification of correctness of both the algorithms as we got the maximum value of 40.

We can also confirm that from the above output that the algo for bottom up and top down also matches with the solution of an exhaustive search which results in correctness of both the algorithms.

From above implementation of both algorithms for verifying the correctness of the results for both top-down, bottom-up approach and exhaustive search results. if both the results are matched then it is the sign of correctness. If both the results don't match, then it is not the sign of correctness.

2. **Performance Comparison on Random Inputs (25 points):** Compare the performance of your two algorithms on random inputs. Generate inputs with random item weights and values (choose your weights to be random integers between 1 and capacity W). Generate plots which show run times with respect to n (for fixed W), and plots which show run times with respect to W (for fixed n). Let n and W be as large as you need them to be to see a performance difference between your two algorithms. Is the performance gap bigger when n is large or when W is large? If there is a difference, try to explain it.

   Deliverable 2: Plots showing the time performance of your algorithm as a function of n and W for each algorithm, and discussion about the reasons for any performance differences between the two algorithms.

Solution:

This below code is to compare the execution time of the top-down and bottom-up approaches of knapsack problem for varying sizes of item sets. This generates random item weights and values, and also measures the execution time for both algorithms, and plots the results to show the performance difference as the size of the item set increases.

Here, we very the number of sizes of items using a random function:

A) *First keep the maximum capacity(W) constant and vary the number of items*:
We vary the number of size of items using a random function:

```
#2nd with constant W
import matplotlib.pyplot as plt
import numpy as np
import random
import time

def constantW():
  n=[x for x in range(10,150,10)]
  W=500
  size=[]
  exectime=[]
  exec_bottom_up=[]
  for i in n:
    weight = [random.randint(1, 15) for _ in range(i)]
    value = [random.randint(1, 25) for _ in range(i)]
    start_time=time.time()
    knapsack_Top_down(weight,value, W,0,{})
    endtime=time.time()
    exectime.append(endtime-start_time)
    start_time1=time.time()
    knapsack_bottom_up(weight,value,i)
    endTime2=time.time()
    exec_bottom_up.append(endTime2-start_time1)
    size.append(i)
  plt.plot(size,exec_bottom_up,label='Bottom up', marker='o')
  plt.plot(size,exectime,label='Top-Down ', marker='o')
  plt.title(f'Running Time vs. size of items)')
  plt.xlabel('Representation Value of n')
  plt.ylabel('Running Time')
  plt.legend()
  plt.show()
```

Fig6. With the constant W

This below code compares the execution time of the top-down and bottom-up approaches to the knapsack problem for varying of knapsack capacities. It generates random item weights and values, measures the execution time for both algorithms, and plots the results to visualize the performance difference as the knapsack capacity increases.

Fixed W Varying N:

For fixed W, the bottom-up approach performs way better than the top-down approach as it has lesser running times.

B) First keep the size of items(n) as constant and vary the number of items:

```python
#constant n
def constantN():
  n=200
  W=[x for x in range(100,1500,50)]
  size=[]
  exectime=[]
  exec_bottom_up=[]
  for i in W:
    weight = [random.randint(1, 150) for i in range(50)]
    value = [random.randint(1, 25) for i in range(50)]
    start_time=time.time()
    knapsack_Top_down(weight,value, i,0,{})
    endtime=time.time()
    exectime.append(endtime-start_time)
    start_time1=time.time()
    knapsack_bottom_up(weight,value,i)
    endTime2=time.time()
    exec_bottom_up.append(endTime2-start_time1)
    size.append(i)
  plt.plot(size,exec_bottom_up,label='Bottom-up', marker='o')
  plt.plot(size,exectime,label='Top-Down ', marker='o')
  plt.title(f'Running Time vs. Capacity')
  plt.xlabel('Capacity W')
  plt.ylabel('Running Time')
  plt.legend()
  plt.show()
constantN();
```

Fig7. With the constant N

The below plot shows the Running time vs Capacity in the below plot. Top down will take much time compared to the Bottom up because of top down is performing above the bottom up and also if top down is increasing simultaneously the bottom up also increases.

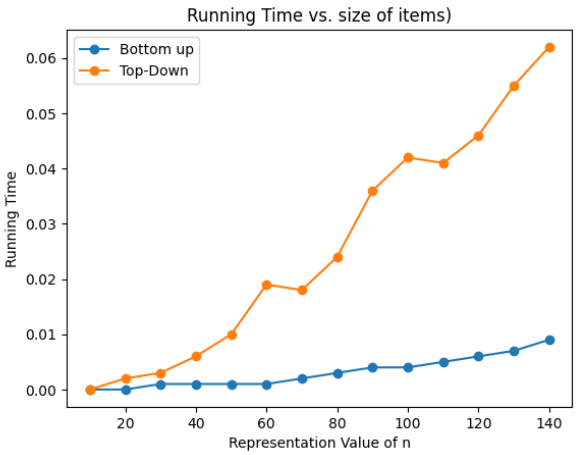

Fig8. Plot between Running time vs Capacity

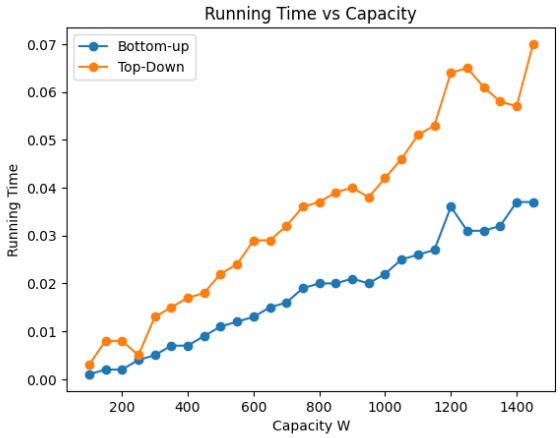

Fig9. Plot between Running time vs Capacity

Performance of Bottom-up vs Top-Down:

The top-down has recursive calls and can have more overhead due to function calls and memoization. While the memoization optimizes by avoiding redundant computations, the overall time complexity may be affected by the recursive structure.

The bottom-up approach typically has a more straightforward time complexity analysis. It systematically fills in a table of subproblem solutions in a specific order, avoiding the overhead of recursive calls. It is often considered more efficient.

Let's see the performance of Fixed N and Fixed W:

Bottom-up vs top down:

Here for fixed n, the execution time increases with increase in W for both top down and bottom-up approaches. We can see that bottom-up approach performs better than top down as it has less running time.

We can also see that always the bottom-up approach performs better and there is a constant time difference between top down and bottom up which is the overhead as top down has recursive function calls.

Fixed W and varying N: We can see that both top downs.

Bottom-up vs top down:

For fixed W, the bottom-up approach performs way better than the top-down approach as it has lesser running times.

Fixed N vs Fixed W:

We can see that the gap between top down and bottom up for fixed N is less comparative to gap between top down and bottom up of fixed W. which implies There is a significant difference in running time between top down and bottom up for Fixed W. Bottom up performs significantly better than top down.

For Fixed N, there is a slight difference between between top down and bottom up.

3. **Performance Comparison on Special Inputs (25 points):** Compare the performance of your algorithms on special inputs. Now, craft inputs where all weights are relatively low (say, select the weights to be random integers between 1 and 1 10). Re-run the comparison between the two algorithms. Has the performance gap changed in any interesting way?
   Deliverable 3: Plots showing the time performance of your algorithm as a function of n and W for each algorithm on special inputs, and discussion about the reasons for any performance differences between the two algorithms.

Solution:

This code analyses the execution time of both the top-down and bottom-up approaches to the knapsack problem for varying numbers of items with only a smallWeights. It also generates random item weights and values, measures the execution time for both algorithms, and then plots the results to show the performance difference as the number of items increases.

```
def constantW_smallWeights():
  n=[x for x in range(1,50)]
  W=30
  size=[]
  exectime=[]
  exec_bottom_up=[]
  for i in n:
    weight = [random.randint(1, 10) for j in range(i)]
    value = [random.randint(1, 25) for j in range(i)]
    start_time=time.time()
    knapsack_Top_down(weight,value,W,0,{})
    endtime=time.time()
    exectime.append(endtime-start_time)
    start_time1=time.time()
    knapsack_bottom_up(weight,value,W)
    endTime2=time.time()
    exec_bottom_up.append(endTime2-start_time1)
    size.append(i)
  plt.plot(size,exec_bottom_up,label='Bottom up')
  plt.plot(size,exectime,label='Top-Down ')
  plt.title(f'Running Time vs size of items (small weights)')
  plt.xlabel('Size of items')
  plt.ylabel('Running Time')
  plt.legend()
  plt.show()
```

Fig10. ConstanW_smallWeights

```
def constantN_smallWeights():
  n=10
  W=[x for x in range(1,1000,20)]
  size=[]
  exectime=[]
  exec_bottom_up=[]
  for i in W:
    weight = [random.randint(1, 10) for k in range(n)]
    value = [random.randint(1, 15) for k in range(n)]
    start_time=time.time()
    knapsack_Top_down(weight,value,i,0,{})
    endtime=time.time()
    exectime.append(endtime-start_time)
    start_time1=time.time()
    knapsack_bottom_up(weight,value,i)
    endTime2=time.time()
    exec_bottom_up.append(endTime2-start_time1)
    size.append(i)
  plt.plot(size,exec_bottom_up,label='Bottom up')
  plt.plot(size,exectime,label='Top-Down ')
  plt.title(f'Running Time vs W (small weights)')
  plt.xlabel('Capacity W')
  plt.ylabel('Running Time')
  plt.legend()
  plt.show()
constantN_smallWeights()
constantW();
```

Fig11. ConstantN_smallWeights

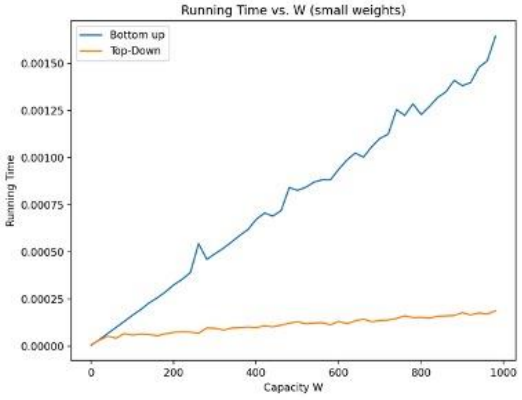

Fig12. Plot between Running time vs W (small weights)

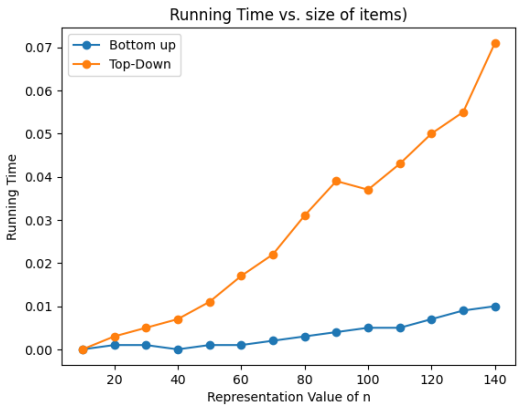

Fig13. Plots between Running time vs Size of items

Fixed W: Top-Down Vs Bottom-Up Approach:

We have seen from the last result that bottom up is better than top down as top down has multiple recursive function calls which causes overhead.

But for small values of w (max 10) Top-down approach has performed as we can see from the above graph for small weights and when the size of the items(n) is less (and small W), the top down has less time complexity than the bottom up.

when the size of n increases the, we can see after a certain point the bottom-up approach performs way better than the top down.

Fixed N Varying W:

Top-Down Vs Bottom-Up Approach:

For small values of W, we can see that top-down approach performs better for Fixed N as it has less running time compared to bottom up in spite of the recursive calls.

Bottom-up approach performs better for large values of W but for small values of weights,

Top-down approach performs better.

4. **Pseudo polynomial-time Illustration (20 points):** Illustrate that this is a pseudo polynomial-time algorithm. The time complexity Θ(nW) is pseudo polynomial: it is polynomial in the value of W, but not the size of the representation of W. Generate a plot which illustrates this characteristic.

   Deliverable 4: A plot with running time on the vertical axis and the size of the representation of W on the horizontal axis which shows that this algorithm is exponential in the size of the representation of W.

Solution:

```
import time
import matplotlib.pyplot as plt

def measure_execution_times_1(values, weights, W):
    start_time_bottom_up = time.time()
    knapsack_bottom_up(values, weights, W)
    end_time_bottom_up = time.time()

    start_time_top_down = time.time()
    knapsack_top_down(values, weights, W)
    end_time_top_down = time.time()

    return end_time_bottom_up - start_time_bottom_up, end_time_top_down - start_time_top_down

def generate_inputs(n, W_size):
    values = [60, 100, 120]
    weights = [10, 20, 30]
    W = 2 ** W_size
    return values, weights, W
```

Fig14. Measure-execution-times and generate inputs

```
def generate_pseudopolynomial_plot():
    W_sizes = list(range(10, 21))
    bottom_up_execution_times = []
    top_down_execution_times = []

    for W_size in W_sizes:
        values, weights, W = generate_inputs(3, W_size)

        bottom_up_execution_time, top_down_execution_time = measure_execution_times_1(values, weights, W)

        bottom_up_execution_times.append(bottom_up_execution_time)
        top_down_execution_times.append(top_down_execution_time)


    print("Bottom-Up Algorithm:")
    for i in range(0, len(W_sizes)):
      print(W_sizes[i], bottom_up_execution_times[i], end=" ")
    print()
    print("Top-Down Algorithm:")
    for i in range(0, len(W_sizes)):
      print(W_sizes[i], top_down_execution_times[i], end=" ")
    plt.plot(W_sizes, bottom_up_execution_times, label='Bottom-Up Algorithm', marker='o')
    plt.plot(W_sizes, top_down_execution_times, label='Top-Down Algorithm', marker='o')

    plt.xlabel('Size of the Representation of W (bits)')
    plt.ylabel('Execution Time (seconds)')
    plt.title('Pseudopolynomial-time Illustration')
    plt.legend()
    plt.show()
```

Fig15. Generating of Pseudo-polynomial plot

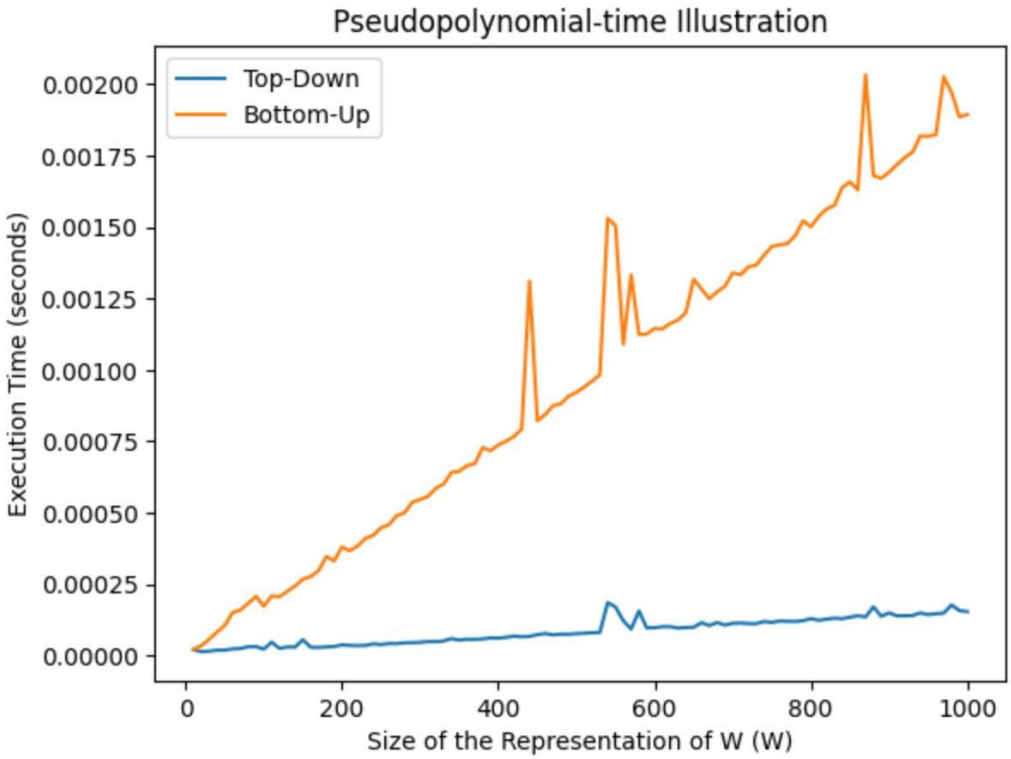

Fig16: Plot for Pseduo-polynomial-time Illustration

For the Knapsack problem, O(nW) is polynomial because the running time grows polynomial with both n and W. It's important to note that the magnitude or values of n and W do not affect the classification of the algorithm's complexity; it's purely based on the input size.

We can see that the graph increases exponentially for bottom-up approach. Hence, we can say it as pseudo polynomial function.