CS333 Project 4

Hasan Can Sevindir

Introduction:

Knapsack problem is an old but still relevant problem in real life that originally tries to calculate to maximize the profit of carrying weighted values with a limited bag capacity. It appears in a wide variety of fields, such as finding the least wasteful way to cut raw materials, selection of investments and portfolios, selection of assets for asset-backed securitization, and generating keys for the Merkle–Hellman and other knapsack cryptosystems.[1] What is the maximum amount of weights with their corresponding values that is possible to carry in a bag with weight carriage limit N to maximize the value/profit? This is the question that knapsack algorithms try to answer. This is also known as knapsack 0-1 problem.

At first glance calculating this seem quite straight forward and not very difficult. However, a more detailed dive quickly proves otherwise. A trivial or greedy algorithm simply is not enough to calculate the result of this problem. One of the most common solutions to this problem uses a dynamic programming approach.

Problem statement:

In this project, it was expected to find a np complete dynamic programming solution to the Knapsack decision problem. Np complete in this case means answering “yes” or “no” to the question whether if it is possible to carry an amount of weight to obtain a minimum amount of given value with the knapsack of given capacity.

The user is expected to enter the number of elements, the weights of the elements, the values of the elements, what the maximum weight that the knapsack can carry and what the minimum profit that is required. It may or may not be possible to actually reach this profit. It may be possible to get a maximum profit N when the minimum required profit that the user entered was M, where M > N.

A successful “Yes” example execution of the program can be as follows:

Input (user inputs are in green):

```
Enter the number of items: 7

Enter weights of items: 2 2 1 4 3 6 1

Enter profit values of items: 30 20 10 30 50 40 10

Enter the maximum weight capacity: 13

Enter the minimum profit required: 140
```

Output:

```
YES

Because the maximum profit possible is 150
```

A successful “No” example execution of the program can be as follows:

Input (user inputs are in green):

```
Enter the number of items: 3

Enter weights of items: 1 3 2

Enter profit values of items: 30 20 10

Enter the maximum weight capacity: 5

Enter the minimum profit required: 60
```

Output:

```
YES

Because the maximum profit possible is 50
```

In the first example, user requested that the minimum profit would be 140. Since it is possible to carry items that their values in the best case will sum up to 150, required profit of 140 is possible to achieve. Therefore, the answer is YES.

In the second example however, user requested that the minimum profit would be 60. Since it is possible to carry items that their values in the best case will sum up to 50, required profit of 60 is not possible to achieve. Thus, the answer is NO.

Solution approach:

There 2 common dynamic problem solutions to knapsack 0-1 problem. One of them uses loops and the other uses recursion to calculate the maximum possible profit (maximum amount of value that can be carried in the knapsack).

1- Solution using loops:

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant
values starting at index 1.

array m[0..n, 0..W];
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    m[i, 0] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

The pseudo code above [2] shows how to solve the 0-1 knapsack problem using dynamic programming, we can define a function m [i, w] which represents the maximum value that can be attained using items 1 through i, with a maximum weight of w. This solution runs in pseudo-polynomial time and all weights (w1 through wn and W) are assumed to be positive integers. To efficiently solve this problem, a table is used to store the results of previous computations.

The implementation of pseudo code above can be as follows in Java:

```java
private int calculateMaxProfit(int capacity) {
   int size = items.size();
   int[][] knapsack = new int[size + 1][capacity + 1];
   for (int i = 1; i <= size; i++) {
      for (int weight = 0; weight <= capacity; weight++) {
         if (items.get(i - 1).weight <= weight)
            knapsack[i][weight] = Math.max(
                  items.get(i - 1).value + knapsack[i -
1][weight - items.get(i - 1).weight],
                  knapsack[i - 1][weight]
            );
         else
            knapsack[i][weight] = knapsack[i - 1][weight];
      }
   }

   return knapsack[size][capacity];
}
```

“items” is an instance of List<KnapsackItem>. A KnapsackItem is defined as follows:

```java
public static class KnapsackItem {
   public final int value;
   public final int weight;

   public KnapsackItem(int value, int weight) {
      this.value = value;
      this.weight = weight;
   }
}
```

It is a utility class for keeping values with their corresponding weights as pairs.

The list of items and calculate functions are defined in a class named KnapsackCalculator.

```
public static class KnapsackCalculator {
   private final List<KnapsackItem> items;
      // . . .
```

What is problematic with this implementation is that we do not need to consider all possible weights when the number and specific items of the selection are fixed. In other words, the previously mentioned program calculates more values than necessary because the weight changes from 0 to W frequently. To improve efficiency, we can implement this method using a recursive approach.

2- Solution using recursion:

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant
values starting at index 1.
Define value[n, W]
Initialize all value[i, j] = -1
Define m:=(i,j)          // Define function m so that it represents the
maximum value we can get under the condition: use first i items, total
weight limit is j
{
    if i == 0 or j <= 0 then:
        value[i, j] = 0
        return
    if (value[i-1, j] == -1) then:      // m[i-1, j] has not been
calculated, we have to call function m
        m(i-1, j)
    if w[i] > j then:                            // item cannot fit in the
bag
        value[i, j] = value[i-1, j]
    else:
        if (value[i-1, j-w[i]] == -1) then:      // m[i-1,j-w[i]] has
not been calculated, we have to call function m
            m(i-1, j-w[i])
        value[i, j] = max(value[i-1,j], value[i-1, j-w[i]] + v[i])
}
```

The pseudo code above [2] shows how to solve the 0-1 knapsack problem using dynamic programming with recursion.

The implementation of pseudo code above can be as follows in Java:

```java
private int calculateMaxProfitRecursively(int capacity, int
itemsSize) {
   int[][] dp = new int[itemsSize + 1][capacity + 1];
   for (int[] array: dp)
      Arrays.fill(array, -1);
   return calculateMaxProfitRecursivelyImpl(capacity, itemsSize,
dp);
}

private int calculateMaxProfitRecursivelyImpl(int capacity, int
itemsSize, int[][] dp) {
   if (itemsSize == 0 || capacity == 0)
      return 0;

   if (dp[itemsSize][capacity] != -1)
      return dp[itemsSize][capacity];

   int lastItemWeight = items.get(itemsSize - 1).weight;
   if (lastItemWeight > capacity) {
      dp[itemsSize][capacity] =
calculateMaxProfitRecursivelyImpl(capacity, itemsSize - 1, dp);
      return dp[itemsSize][capacity];
   }
   int lastItemValue = items.get(itemsSize - 1).value;
   dp[itemsSize][capacity] = Math.max(
         lastItemValue + calculateMaxProfitRecursivelyImpl(capacity
- lastItemWeight, itemsSize - 1, dp),
         calculateMaxProfitRecursivelyImpl(capacity, itemsSize - 1,
dp)
   );
   return dp[itemsSize][capacity];
}
```

Time Complexity Analysis:

For the previous algorithm that uses loops, time complexity ranges from O(n*W) (simply because of the nested loops) to O(2^n) because of the redundancy discussed above.

With the recursive approach, a new table value[n, W] is defined and used as a condition for branching in recursion. Without this table, this algorithm would have had O(2^n) time complexity. It can be shown as below:

```
                         K(n, W)
                         K(3, 2)
                     /             \
                   /                 \
            K(2, 2)                    K(2, 1)
          /        \                   /    \
        /            \               /        \
    K(1, 2)        K(1, 1)        K(1, 1)       K(1, 0)
     /  \           /   \           /   \
   /      \       /       \       /       \
K(0, 2)  K(0, 1)  K(0, 1)  K(0, 0)  K(0, 1)   K(0, 0)
```

where K() represents the knapsack calculation function.

However, with the help of the value table in the recursive approach, the time complexity becomes O(n*W) as redundant calculations that are seen in the figure above are avoided.

In my code, I implemented both of the cases and depending on which algorithm the code is desired to be run with, simply alternating the comment on the following lines are sufficient:

```java
public KnapsackCalculator(int capacity, List<KnapsackItem> items) {
   this.items = items;
   // result = calculateMaxProfit(capacity);
   result = calculateMaxProfitRecursively(capacity, items.size());
}
```

Commenting or uncommenting both of the lines at the same will result in a compile error, so only 1 of them should be commented and uncommented at once. calculateMaxProfit() function uses the loop based algorithm and calculateMaxProfitRecursively() uses the recursive algorithm.

Conclusion:

I have discussed strong and weak sides, time complexities and implementation details of algorithms about dynamic programming solutions to knapsack problem in this paper. After being able to calculate the knapsack problem and the max possible profit, it is trivial to check is minimum desired profit is possible to be achieved or not. This is now an NP complete yes or no answering program to knapsack decision problem.