COM361 Assignment 3
User Name: luqing
Student ID: 300363602

1 Solve the 0-1 knapsack problem using dynamic programming

1.1 Provide an implementation of your algorithm (say in Java or whichever language you prefer subject to tutor's approval);

I used Java to implement a dynamic programming-based algorithm to address the 0-1 knapsack problem. The proposed algorithm has been tested on IntelliJ and has produced correct results for all the test samples. The source code is stored in folder $knapsack01DP$.

1.2 A description of your algorithm in pseudocode and a proof (or convincing argument) of why it is correct;

The 0-1 knapsack problem provides n items, i.e., ($Item_1$ ... $Item_n$). Each item has a weight and a value, say ($Weight_n$, $Value_n$) $\in Item_n$. The maximal capacity of a knapsack is W. The question is to select i items from n items, in which the total weight of selected items is smaller or equal to W, and the maximum total value is achieved. Note that a 0-1 knapsack problem does not allow any duplicate item.

I split the target problem into sub-problems V[i, C], which represents the maximum total value of the first i items (1$\leq i \leq n$), where we have completed the selection process for these items, and the selected items' total weight is no larger than C (1$\leq C \leq W$).

For any ith item, it can either be selected or be unselected. If I do not put the ith item into my knapsack, the sub-problem is V[i-1, C]. Oppositely, if I add the ith item into my knapsack, my bag's capacity will reduce, i.e. the sub-problem is V[i-1, C-$Weight_i$] (Shown in Fig. 1). Thus, the recurrence is shown in equation 1. Note that as the assignment requires me to output the selected items, thus, I create an object bag to represent the result for each V[i, C]. Each bag stores the value and the selected items in each V[i, C]. The pseudocode is shown in Algorithm 1 and 2. The final solution of the proposed dynamic programming is stored in V[n, W].

$$V[i,C]\begin{cases} if \quad i=0: & 0 \\ if \quad C=0: & 0 \\ if \quad Weight_i > C: & V[i-i,C] \\ otherwise: & max(V[i-1,C], V[i-1,C-Weight_i]+Value_i) \end{cases} \tag{1}$$

Here is the proof for the proposed algorithm:

Inputs:

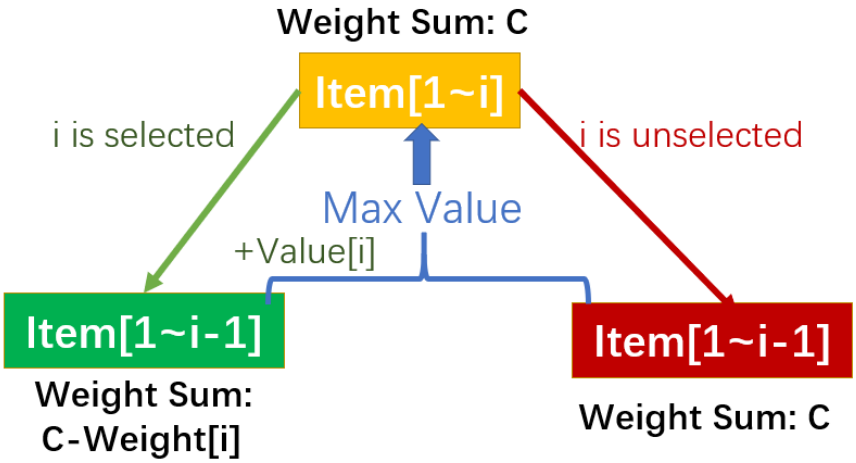

Figure 1: Graph shows the samples of the sub problems, i.e. V[i, C] = max(V[i-1, C] , V [i-1, C-$Weight_i$] +$Value_i$).

1. **n**: number of items;
2. **W**: capacity of the given knapsack;
3. **Items**: a collection of items.

Output:
An (n+1) x (W+1) array: Each element in this array is a Bag object. A Bag object contains a list of the selected items, and the total value of these items.

Theorem (Assumption):
V[n][W] records the final result, i.e., the item combination that reaches the maximum total value with the total weight less than the given W.

Proof (Requirement):
By induction on V[i][C], i∈[0,n] and C ∈ [0,W].

Base case:
When i=0, V[0][C].value =0, V[0][C].Items=[]: when i is 0, it means there is no items; thus, no item can be selected, and the value is 0 in this case. When C=0, V[i][0].value =0, V[i][0].Items=[]. This is because W=0 indicates that the given knapsack cannot have any items. Thus, the total value of this case must be 0.

Induction hypothesis:
V[n][W] correctly records the best item combination and maximum sum value, that the sum weight is lower than the given W.

Induction step: prove the induction hypothesis also suits n+1

1. Suppose the answer in V[n][W] is correct, and we know the value and weight of $item_{n+1}$
2. Regarding $item_{n+1}$, there are two cases: $item_{n+1}$ is selected or $item_{n+1}$ is NOT selected
3. $item_{n+1}$ is selected, then the sum value of V[n+1][W].value = V[n][W].value + $item_{n+1}$.value. According to the induction hypothesis, V[n][W].value is correct. Hence, the obtained result for the V[n+1][W] is also correct.

Algorithm 1: Dynamic programming for the 0-1 knapsack problem

Input: n: number of items; **W**: capacity of the given knapsack;
Items: a set of items
Output: Solution: a knapsack with selected item that reach the maximum total value

```
V[n+1][W+1] ← an empty array, each element is a Bag object, bag.value←0.
foreach i ∈ [1,n] do
    foreach C ∈ [1,W] do
        if Item_i.Weight > C then
            V[i+1][C] ← V[i][C]
        end
        else
            V[i+1][C]← Max(V[i][C], V[i][C-Item_i.Weight], Item_i)
        end
    end
end
Solution ← V[n][W]
```

Algorithm 2: Max returns the better knapsack

Input: Bag_a:A bag; Bag_b: A bag; $Item_i$: an item
Output: result: a selected knapsack

```
if Bag_a.Value > Bag_b.Value+Item_i.Value then
    Return Bag_a
end
else
    Bag_b.Value ← Bag_b.Value+Item_i.Value
    Bag_b.Items.Add(Item_i.Name)
    Return Bag_b
end
```

4. $item_{n+1}$ cannot be selected, then the sum value of V[n+1][W].value is equal to the V[n][W].value. According to the induction hypothesis, this is also correct.

5. Therefore, the proposed induction hypothesis correctly shows the case of n+1, as the answers for both possible cases are correct.

1.3 A test case generator to be able to test your algorithm and a description of how the generator works.

I code a class named $RandomSample$ for testing. The most important function is GenerateSamples(int (n) - number of items, int (v) - maximum value for items, int (w) - maximum weight for items). This function automatically generates n items, with each item's weight less than the given w, and each item's value less than the given v.

Another important function is GenerateSamples (int[] weight, int [] value). This function returns an item list based the given values. For example, if the input is weight=[0,1], value=[1,2], the output of this function is [item(w=0,v=1), item (w=1,v=2)].

Put the item list and an int type W into Knap01's RunAlgorithm function, then it starts testing. Samples and other detailed information are in the main function in the Runner class.

1.4 Plots of your algorithm using an appropriate barometer and a conclusion of what your implementation's complexity is (include a discussion of the expected complexity and the one you get using your test case generator and barometers).

Table 1: Table for barometer results from n items and a given W.

n	5	10	15	20	25
W	5	10	15	20	25
barometer	25	100	225	400	625

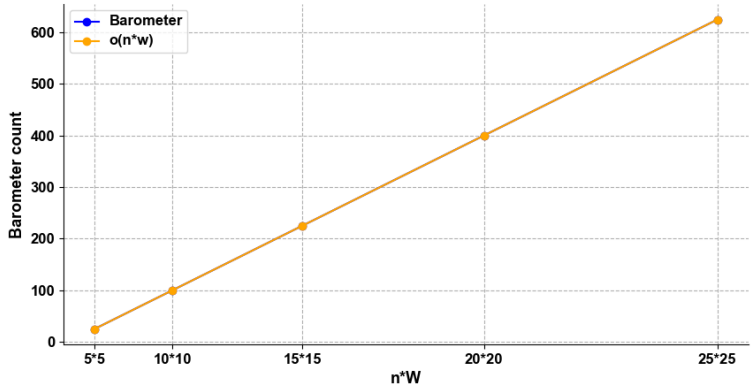

Figure 2: This graph shows the relationship between execution times and different numbers of n*W. Blue line: barometer's estimated complexity; orange line: theorem complexity.

For a given problem that contains n items and W capacity, the proposed algorithm requires an (n+1)*(W+1) size array. Thus, the space complexity for the proposed algorithm is n*W. According to the pseudocode, the proposed algorithm only has one nested loop from line 2 to 11. Thus, the proposed algorithm's expected complexity (theorem complexity) is θ (n*W). I increased the barometer whenever an element of V[n+1][W+1] is updated. The barometer results (estimated complexity) of the test cases are shown in Table 1. Furthermore, the graph for both expected complexity and barometer results is shown in Fig. 2.

All the elements of V[n][w] need to be calculated to ascertain the best item selection. This claim is supported by Fig. 2, where graphs for expected complexity and barometer results are completely overlapped.

1.5 Screenshot evidence that shows your program can work properly and produce desirable results.

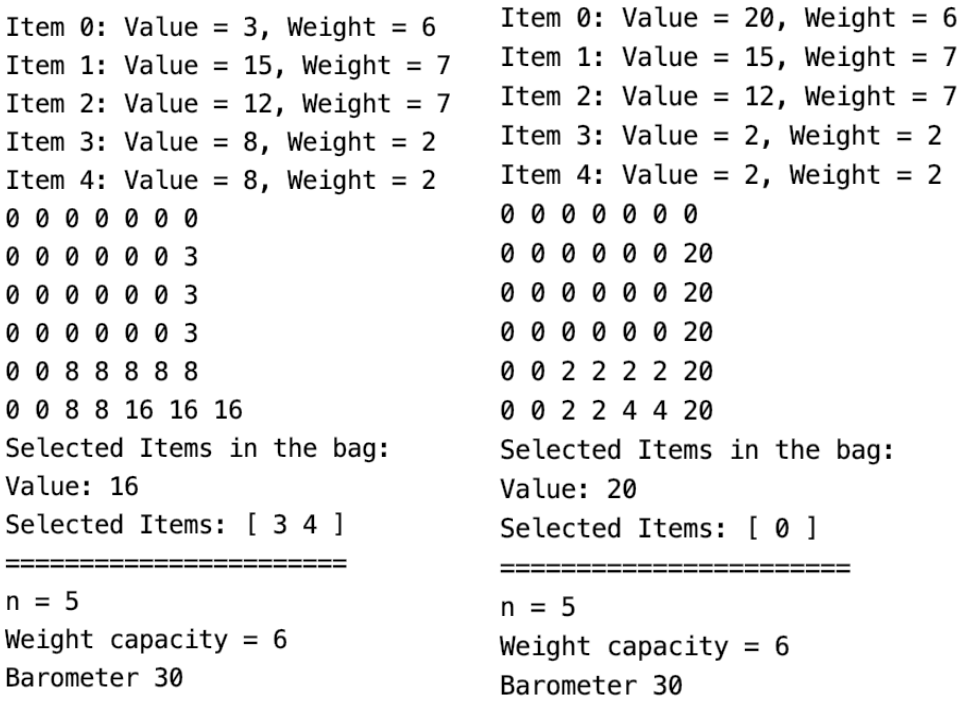

Figure 3: The two screenshots show the results of the proposed algorithm.The results demonstrate the proposed algorithm is correct. In the left case, 5 items are given: [Item0: v=3, w=6], [Item1: v=15, w=7], [Item2: v=12, w=7], [Item3: v=8, w=2], [Item4: v=8, w=2]. The given W is 6. Thus, the expected result is [3,4] with a maximum value of 16. In the right case, the given 5 items are [Item0: v=20, w=6], [Item1: v=15, w=7], [Item2: v=12, w=7], [Item3: v=2, w=2], [Item4: v=2, w=2]. The given W is 6. Thus the expected result has the value of 20 and selects $item_0$.

2 Solve the 0-N knapsack problem using brute force

2.1 Provide an implementation of your algorithm (say in Java or whichever language you prefer subject to tutor's approval);

I used Java to implement a brute force-based algorithm to address the 0-N knapsack problem. The proposed algorithm has been tested on IntelliJ and has produced correct results for all the test samples. The source code is stored in the folder $knapsack0NBF$.

2.2 A description of your algorithm in pseudocode and a proof (or convincing argument) of why it is correct;

It is practicable to translate a bounded knapsack problem into a 0-1 knapsack problem. For example, a bounded knapsack problem that contains two items like [$item_0$(num=2), $item_1$(num=2)]

is equal to a 0-1 knapsack problem, which has four items, i.e. [$item_{01}$, $item_{02}$, $item_{11}$, $item_{12}$]. Now the number of items is equal to $\sum n$, i.e. $\sum n$= $item_1$.number+ ... $item_n$.number.

The basic idea of this part is to utilize a boolean-based representing format to represent the selection state of each given item, i.e., 0 means selected, 1 means unselected. For example, if a problem's $\sum n$ is 6, and we select all the items, this case can be represented by 111111. Oppositely, if we do not select any item, this case can be represented by 000000. Thus, for any n items problem, there exist $2^{\sum n}$ situations. Algorithm 3 describes how to get all the possible 2^n situations. Similar to the solution of the previous question, here I utilize a Bag object to record the result for each possible strategy. A bag records the sum value and sum weight for each strategy, i.e., bag.value and bag.weight. Algorithm 4 shows how to calculate the sum value and sum weight for each possible strategy. Algorithm 5 finds the best strategy from Algorithm 4's result.

Algorithm 3: Find all the possible strategies

Input: m: the value of $\sum n$;
Output: S[2^m]**[m]**: a list that contains all the possible strategies;

```
S[2^m][m] ← 0 for all the elements
for i ∈ [0,2^m] do
    value ← i
    divisor ← 2^m
    for j ∈ [0,m] do
        divisor ← divisor/2
        if value ≥ divisor then
            S[i][j] ← 1
            value ← value - divisor
        end
    end
end
```

Algorithm 4: Calculate result for each possible strategy

Input: m: the value of $\sum n$;
S[2^m]**[m]**: a list contains all possible strategies obtained from the Algorithm 3
items: a list that contains m items
Output: R[2^m]: a list that contains the results for all the possbile strategies

```
R[2^m] ← initialize the list, all elements are bag instances, bag.value, bag.weight ← 0.
for i ∈ [0,2^m] do
    for j ∈ [0,m] do
        if S[i][j] is 1 then
            R[i].weight+= item[i].weight
            R[i].value+= item[i].value
        end
    end
end
```

Proof: Input: a flattened item list, i.e., n items become $\sum n$ items and W, which is the capacity of the given knapsack.
Output: the best strategy.

Algorithm 5: find the best strategy

Input: m: the value of $\sum n$;
R[2^m]: a list that contains the results for all possible strategies;
S[2^m]**[m]**: a list contains all possible strategies that obtained from the Algorithm 3
W: the maximum capacity of the given knapsack;
Output: Best: the best strategy

```
BestId ← -1
BestValue ← -1
for i ∈ [0,2^m] do
    if R[i].weight ≤ W And BestValue ≤ R[i].value then
        BestValue ← R[i].value
        BestId ← i
    end
end
Return S[i], R[i]
```

Base case: when W=0, the strategy 00..0 means nothing can be selected.

Induction: Assuming $\sum n$ is equal to m, for any m items problem, there exist 2^m strategies, and the best case must belong to these 2^m strategies. Because the proposed algorithm utilizes brute force to get all results for all the possible strategies, the selected best strategy must be the answer for the target question, i.e., selecting i items from given m items can achieve the largest total value together with the sum weight less than the given W.

2.3 A test case generator to be able to test your algorithm (this can then be reused for the rest of this assignment) and a description of how the generator works.

I code a class named $RandomSample$ for testing. The most important function is GenerateSamples(int (n)-number of items, int-(v) maximum value for items, int (w)-maximum weight for items, int (s)-maximum size for items). This function automatically produces n items, with each item's weight less than the given w, the value less than the given v, and size between [1, s].

Another important function is GenerateSamples (int[] weight, int [] value, int[] size). This function returns an item list with the given value.For example, if the input is weight=[0,1], value=[1,2], size=[2,3], the output of this function is [item(w=0, v=1, s=2), item (w=1, v=2, s=3)].

Put the item list and an int type W into bruteForce class's FindStrategy function, then it starts testing. Samples and other details are in the Runner class's main function.

2.4 Plots of your algorithm using an appropriate barometer and a conclusion of what your implementation's complexity is (include a discussion of the expected complexity and the one you get using your test case generator and barometers).

Assuming after flatten the 0-N knapsack problem to the 0-1 knapsack problem, the $\sum n$ is equal to m. The space complexity of the proposed algorithm is 2^m*m + 2^m, which is approximate to 2^m*m.

The theorem complexity of algorithm 3, 4, and 5 respectively are O(2^m*m), O(2^m*m) and O(2^m). Thus, the complexity of the proposed brute force algorithm is θ(2^m*m). I increase the barometer whenever the S[2^m][m] is updated or being used for computation. The results are shown in Table 2.

Table 2: Table for barometer results from $\sum n$ (m) items.

m	6	7	8	9	10
barometer	768	1792	4096	9216	20480

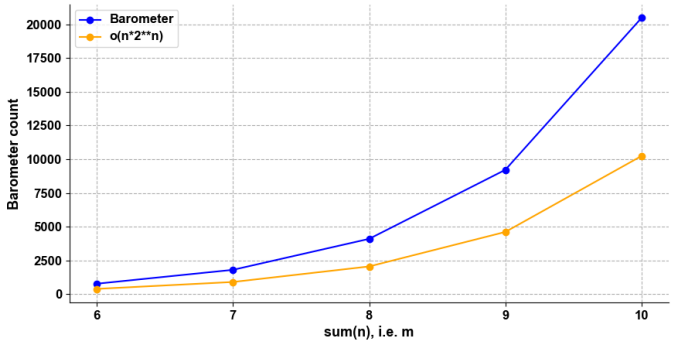

Figure 4: Graph shows the relationship between execution times and different numbers of m ($\sum n$). Blue line: the barometer's estimated complexity; orange line: the theorem complexity.

Fig. 4 shows the barometer's estimated complexity and theorem complexity for the proposed algorithm. Barometer's estimated complexity is equal to 2* 2^m*m, whereas the theorem complexity is an approximate value, which is 2^m*m. That is the reason why the theorem complexity's graph is under the barometer's estimated complexity's graph.

2.5 Screenshot evidence that shows your program can work properly and produce desirable results.

```
Item0: Value = 15 Weight = 7 Number = 1
Item0: Value = 15 Weight = 7 Number = 1
Item1: Value = 12 Weight = 6 Number = 1
Item2: Value = 8 Weight = 2 Number = 1
Item2: Value = 8 Weight = 2 Number = 1
given Weight capacity: 7
Select Items: Item2 Item2  | Total Weight: 4  | Total Value: 16
Barometer 320
```

Figure 5: This shows the results achieved by the proposed algorithm. The screenshot demonstrates the proposed algorithm is correct. This 0-N knapsack problem has three items: [Item0: V=15, W=7, S=2], [Item1: V=12, W=6, S=1], and [Item2: V=8, W=2, S=2]. The given W is 7. Thus, the expected solution for this problem is to select two $item_2$, and the maximum total value is expected to be 16.

3 Solve the 0-N knapsack problem using dynamic programming

3.1 Provide an implementation of your algorithm (say in Java or whichever language you prefer subject to tutor's approval);

I used Java to implement a dynamic programming-based algorithm to address the 0-N knapsack problem. The proposed algorithm has been tested in IntelliJ and has produced correct results for all the testing samples. The proposed algorithm is in the folder $knapsack0NDP1$. Furthermore, I also did an alternative solution for the 0-N knapsack problem, the source code of which is in the folder $knapsack0NDP2$.

3.2 A description of your algorithm in pseudocode and a proof (or convincing argument) of why it is correct

Algorithm 6: Dynamic programming for the 0-N knapsack problem

Input: m: the $\sum n$; **W**: capacity of the given knapsack;
Items: a set of items
Output: Solution: a knapsack with selected items that reach the maximum sum value

```
V[m+1][W+1] ← an empty array, each element is a Bag object, bag.value←0.
foreach i ∈ [1,m] do
    foreach C ∈ [1,W] do
        if Item_i.Weight > C then
            V[i+1][C] ← V[i][C]
        end
        else
            V[i+1][C]← Max(V[i][C], V[i][C-Item_i.Weight], Item_i)
        end
    end
end
Solution ← V[m][W]
```

It is practicable to translate a bounded knapsack problem into a 0-1 knapsack problem with flattening the given items, e.g. [$item_0$(num=2),$item_1$(num=2)] becomes [$item_{01}$, $item_{02}$, $item_{11}$, $item_{12}$] after the flattening process.

After translating the 0-N knapsack problem into the 0-1 knapsack problem, the target problem can be addressed by the proposed algorithm 6. For a given n items problem, assume $\sum n$ is equal to m, i.e., m= $item_1$.number+...$item_1$.number. Please note that the employed Max function is the same as Algorithm 2. The recurrence of this algorithm is shown in Equation 1.

Here is the proof for the proposed algorithm:

Inputs:

1. **m**: the number of the flattened given n items, i.e., $\sum n$;
2. **W**: capacity of the given knapsack;
3. **Items**: a set of flattened items.

Output:

An (m+1)*(W+1) array: each element in this array is a Bag object. A Bag object contains a list of the selected items and the sum value of these items.

Theorem (Assumption):

V[m][W] records the final result, i.e., the item combination that reaches the maximum sum value with the total weight less than the given W.

Proof (Requirement):

By induction on V[i][C], i∈[0, m] and C ∈ [0, W].

Base case:

When i=0, V[0][C].value =0, V[0][C].Items=[]: when i is 0, it means there is no items. Thus, there is no item to be selected, and the value in this case is 0. When C=0, V[i][0].value =0, V[i][0].Items=[]. This is because W=0 indicates the given knapsack cannot have any items. Thus, the sum value must be 0 in this case.

Induction hypothesis:

V[n][W] correct records the best item combination and maximum sum value, and the sum weight is lower than the given W.

Induction step: prove the induction hypothesis also suits m+1

1. Suppose the answer in V[m][W] is correct, and we know the value, weight and number of given $item_{m+1}$
2. Regarding $item_{m+1}$, there are two cases: $item_{m+1}$ is selected or $item_{m+1}$ is NOT selected
3. $item_{i+1}$ is selected: the sum value of V[m+1][W].value = V[m][W].value + $item_{i+1}$.value. According to the induction hypothesis, V[m][W].value is correct, then the obtained result for the V[m+1][W] is also correct.
4. $item_{m+1}$ cannot be selected: the sum value of V[m+1][W].value is equal to the V[m][W].value. According to the induction hypothesis, this is also correct.
5. Therefore, the proposed induction hypothesis correctly answers the case of m+1, as the answers for both possible cases are correct.

The alternative version for this part does not flatten the give items. The recurrence is shown in equation 2. Note: 0 ≤ k ≤ $item_i$.number and 0 ≤ k*v[i] ≤ C. Algorithm 7 describes the pseudocode of the alternative. The prove of the alternative is similar to the proof of the proposed algorithm. Hence, I do not repeat it here.

$$V[i, C] = \{max(V[i-1][C - k * Item_i.weight] + k * Item_i.value)\} \tag{2}$$

3.3 A test case generator to be able to test your algorithm (this can then be reused for the rest of this assignment) and a description of how the generator works.

I code a class named RandomSample for testing. The most important function is GenerateSamples(int (n)-number of items, int (v)-maximum value for items, int (w)-maximum weight for items, int (s)-maximum size for items). This function automatically generates n items, with each item's weight less than the given w, the value less than the given v, size between [1, s].

Another important function is GenerateSamples (int[] weight, int [] value, int[] size). This function returns an item list with the given value, e.g., if the input is weight=[0, 1], value=[1, 2], size=[2, 3], the output of this function is [item(w=0, v=1, s=2), item (w=1, v=2, s=3)].

Put the item list and an int type W into DynamicProgramming class's Nknapsack function, then it starts testing. Samples and other details are in the Runner class's main function.

Algorithm 7: Dynamic programming for the 0-N knapsack problem

Input: n: the number of items; **W**: capacity of the given knapsack;
Items: a set of items
Output: Solution:the maximum sum value can achieved

```
V[W+1] ← an empry array for i ∈ [0,n] do
    for j ∈ [W,V[i]] do
        for k ∈ [0,Item[i].number] do
            if j ≥ k*Item[i].value then
                max(V[j],V[j-K*Item[i].weight]+k*Item[i].value)
            end
        end
    end
end
solution ← V[W]
```

The source code of the alternative version is in the DP class. You can run the alternative class's main function to do the test. Please note that you can set the value of int[] value, int[] weight, and int[] size to allow different samples' testing.

3.4 Plots of your algorithm using an appropriate barometer and a conclusion of what your implementation's complexity is (include a discussion of the expected complexity and the one you get using your test case generator and barometers).

Table 3: Table for barometer results from $\sum n$ (m) items and a given W.

m	5	10	15	20	25
W	5	10	15	20	25
barometer	25	100	225	400	625

Table 4: Table for alternative version's barometer results from $\sum$ n items and a given W.

$\sum$ n	10	20	30	40	50
W	10	20	30	40	50
barometer	51	160	301	692	1115

For a given problem that contains n items and W capacity, assume after flattening the n items, there exist m items, i.e., $\sum n$ = m. The proposed algorithm builds an (m+1)*(W+1) size array. This indicates the space complexity is m*W. The theorem complexity of the proposed algorithm is θ (m*W) according to Algorithm 6. I increased the barometer whenever an element of V[m+1][W+1] is updated. The barometer results are shown in Table 3.

Fig. 6 describes the theorem complexity and barometer's estimated complexity. The graphs for both complexities are completely overlapped. This is because all the elements of V[m][w]

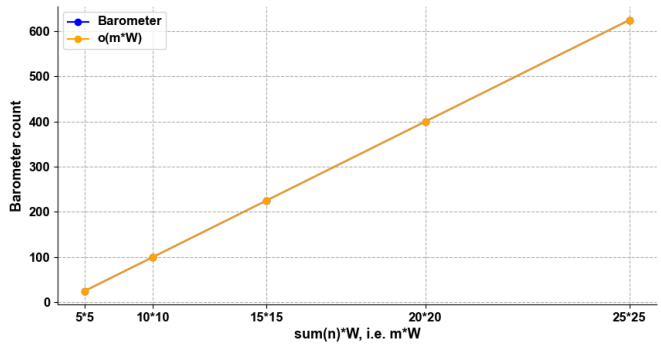

Figure 6: Graph shows the relationship between execution times and different numbers of m ($\sum n$). Blue line: the barometer's estimated complexity; orange line: the theorem complexity.

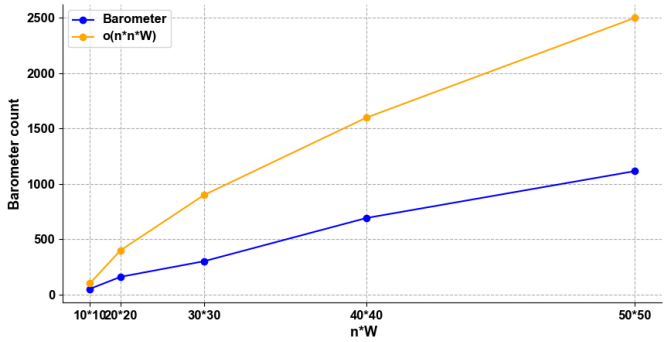

Figure 7: Graph shows the relationship between execution times and different numbers of n*W. Blue line: the barometer's estimated complexity; orange line: the theorem complexity.

need to be calculated to ascertain the best item selection.

The alternative version's theorem complexity is θ ($\sum$ n* $\sum$ n*w). However, in practice, because it is hard for item's distribution to be the worst case, the barometer estimated complexity (shown in Table 4) is much better than the theorem complexity, shown in Fig. 7.

3.5 Screenshot evidence that shows your program can work properly and produce desirable results.

Figure 8 shows the results achieved by the proposed algorithm (please see the next page).

4 Solve the 0-N knapsack problem using graph search

4.1 Provide an implementation of your algorithm (say in Java or whichever language you prefer subject to tutor's approval);

I used Java to implement a graph-search-based algorithm to address the 0-N knapsack problem. The proposed algorithm has been tested in IntelliJ and has produced correct results for all

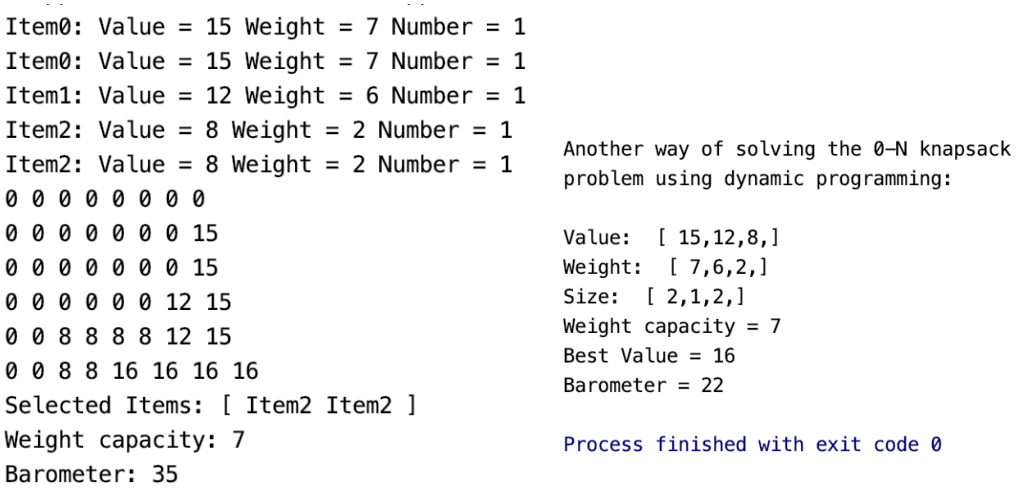

Figure 8: The two cases show the results from two different ways (flatten version, and an alternative version) that I implemented. They demonstrate the proposed algorithm is correct. This 0-N knapsack problem has three items: [0: V=15, W=7, S=2], [1: V=12, W=6, S=1], and [2: V=8, W=2, S=2]. The given W is 7. Thus, the expected solution for this problem is to select two $item_2$, and the maximum sum value is expected to be 16.

the tested samples. The proposed algorithm is named $knapsack0NGS$.

4.2 A description of your algorithm in pseudocode and a proof (or convincing argument) of why it is correct

The basic idea of utilizing graph search to address the 0-N knapsack problem is: find the best node of a node-graph, which represents all the possible strategies of selecting items. Assume a problem has n items, a knapsack has a given W capacity, and an item has three parameters, i.e., value, weight, and number. A node should record the number of each selected item. Then, it is possible to calculate the sum weight and the sum value of the selected items for each vertex. Start node selects nothing. Hence, both sum weight and sum value are zero. The edge tells us how the vertex is linked. Thus, it is practicable to find neighbours by searching a given vertex's edges. Each edge means a new item is selected. For example, assuming a problem has three items, say i, j, and k, the possible edges are (i+1, j, k), (i, j+1, k), and(i, j, k+1). We need to find a node with a sum weight $\leq$ W that maximizes the value. The proposed algorithm is shown in Algorithm 8.

proof for the proposed algorithm:

1. **n**: the number of the items;
2. **W**: capacity of the given knapsack;
3. **Items**: a set of items.

Output:

The "best" node: a node that has the maximum total value together with a sum weight that is less than given W.

Theorem:

Algorithm 8: graph search for the 0-N knapsack problem

Input: n: the number of items; **W**: capacity of the given knapsack
Items: a set of items
Output: best:the best node that reach the maximum value with a weight less than W

```
1  best ← empty node
2  fringe ← empty stack
3  start ← the initialize node, i.e. [0..0] for all the items
4  while fringe is not empty do
5      Current ← fringe.pop()
6      if Current is not already visited then
7          neighour← empty list
8          for i ∈ [0,n] do
9              if Item[i].number ≥ (current.item[i].selected+1) then
10                 find node temp, and set its' value, weight. and selected items.
11                 if temp is not already visit then
12                     fringe.push(temp)
13                 end
14                 if acceptance(temp) then
15                     best ← temp
16                 end
17             end
18         end
19     end
20 end
```

The proposed graph search visits every node reachable from the given start node, and these nodes have represented all the strategies for the given problem.

Proof: by induction on for the given n items, the reachable nodes have represented all the possible strategies.

Base case: 0 items, then node=[0], node.weight=0, node.value=0;
Induction hypothesis: for the given n items, all possible strategies have been represented.
Induction step: prove the induction hypothesis also suits n+1

1. Suppose for the n items case, we have got the correct graph.
2. Regarding the n+1 case, the additional item can be a member of existing items, or can be a completely new item.
3. if n+1 is a new member of existing items, the line 9 in pseudocode ensures all the new nodes to be visitable, i.e., consider the case of any item.number+1.
4. if n+1 is a new item, the line 8 in pseudocode ensures all the new nodes are visitable, i.e., consider the case of item+1.
5. $item_{m+1}$ cannot be selected: the sum value of V[m+1][W].value is equal to the V[m][W].value. According to the induction hypothesis, this is also correct.
6. The proposed induction hypothesis can visit all the new nodes for the case of n+1.

The graph has exhausted all the possible strategies of the item selection. Thus, the best individual node is the solution to the target problem.

4.3 Plots of your algorithm using an appropriate barometer and a conclusion of what your implementation's complexity is (include a discussion of the expected complexity and the one you get using your test case generator and barometers).

Assume a problem has n items: $item_1..itemn$, $\sum$ n = $item_1$.number+...$item_n$.number. Then, the proposed algorithm's theorem complexity is equal to the number of nodes, which is equal to ($item_1$.number+1) * ($item_2$.number+1) ... * ($item_n$.number+1). This theorem complexity is approximate to the number of all possible strategies, i.e., $2^{\sum n}$ (the same as the complexity of the brute force algorithm). This theorem complexity may also equal the worst case of dynamic programming. The barometer's experimented complexity is recorded in Table 5.

Table 5: Table for alternative version's barometer results from n items and a given W.

$\sum n$	10	20	30	40	50
barometer	80	280	920	2645	4330

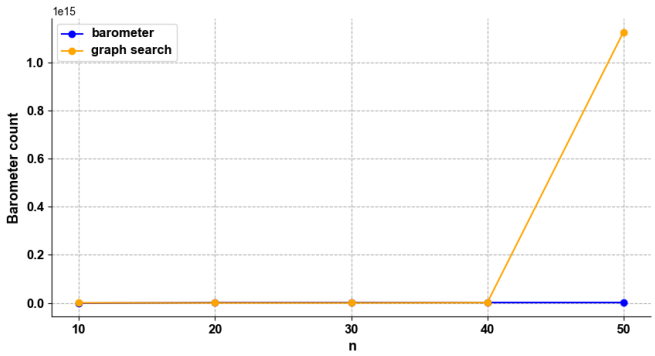

Figure 9: Graph shows the relationship between execution times and different numbers of $\sum n$. Blue line: the barometer's estimated complexity; orange line: the theorem complexity.

The given items combination does not always lead to the worse case. Thus, the barometer's expected complexity is better than the expected complexity. This is supported by Fig. 9.

4.4 Screenshot evidence that shows your program can work properly and produce desirable results.

Figure 10 shows the results achieved by the proposed algorithm.

```
Item0: Value = 15 Weight = 7 Number = 2
Item1: Value = 12 Weight = 6 Number = 1
Item2: Value = 8 Weight = 2 Number = 2
Weight capacity: 7
002:  Weight = 4 Value = 16
Number of each item (same order as above): [ 0 0 2 ]
Barometer: 18
```

Figure 10: Here is an example to show the proposed algorithm is correct. This 0-N knapsack problem has three items: [Item0: V=15, W=7, S=2], [Item01: V=12, W=6, S=1], and [Item02: V=8, W=2, S=2]. The given W is 7. Thus, the expected solution for this problem is to select two $item_2$, and the maximum sum value is expected to be 16.

5 Challenge: improve your graph search algorithm

5.1 Improve your graph search algorithm to perform clearly faster than the dynamic programming algorithm. Provide plots of your improved algorithm using an appropriate barometer to experimentally show the improvement in performance.

Table 6: Table for improved graph search algorithm's barometer results from $\sum$ n items and a given W.

$\sum$ n	10	20	30	40	50
W	10	20	30	40	50
barometer	14	29	69	89	209

Table 7: Table for dynamic programming algorithm's barometer results from $\sum$ n items and a given W.

$\sum$ n	10	20	30	40	50
W	10	20	30	40	50
barometer	51	160	301	692	1115

The source code is in the folder $knapsack0NGSImprove$. Table 6 and Table 7 respectively record the barometer results for the graph search algorithm and dynamic algorithm (the alternative version). Regarding graph search, I set the barometer whenever a node's value is calculated. Regarding the dynamic programming algorithm, the barometer increases whenever the program calculates the value. Furthermore, the collected results are obtained from the same test samples. Thus, I set the barometer appropriately for both algorithms. The results of the two tables demonstrate that the improved graph search is much faster than the dynamic algorithm. This claim is also supported by Fig. 11.

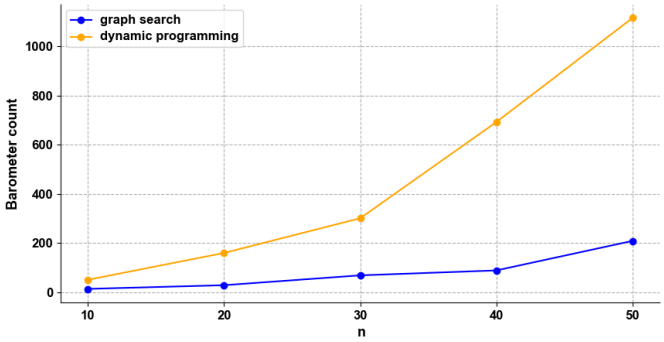

Figure 11: Graph shows the relationship between execution times and different numbers of $\sum n$. Blue line: the graph search; orange line: the dynamic programming.

5.2 A description of why your improved graph search algorithm can perform faster.

I used three strategies to improve my graph search algorithm:

1. Prune unpromising path like sum weight is greater than the given W.
2. Use the greedy algorithms' results as the bound to prune path that clearly cannot produce a better solution.
3. Use the Best-First search to increase the opportunity to find the best node earlier.

The improved graph search algorithm performs faster because it prunes the identified unpromising path by taking advantage of the above three strategies.

5.3 Screenshot evidence that shows your program can work properly and produce desirable results.

Figure 12 shows the results achieved by the proposed algorithm.

```
Item2 : Weight = 2.0 Value = 8.0
Item0 : Weight = 7.0 Value = 15.0
Item1 : Weight = 6.0 Value = 12.0
Weight capacity: 7.0
200:  Weight = 4.0 Value = 16.0
Number of each item (same order as above): [ 2 0 0 ]
Barometer: 8
```

Figure 12: Here is an example to demonstrate the proposed algorithm is correct. This 0-N knapsack problem has three items, i.e. [Item0: V=15, W=7, S=2], [Item1: V=12, W=6, S=1], and [Item2: V=8, W=2, S=2]. The given W is 7. Thus, the expected solution for this problem is to select two $item_2$, and the maximum sum value is expected to be 16.

6 Theoretical questions

6.1 Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.)

Algorithm 9: Report whether a given undirected graph contain a cycle : cycleSearch

```
   Input: a node N
   Output: 1 if a cycle is found
 1 N.visited ← true
 2 for N′ ∈ [neighbours(N,edge)] do
 3 |   if N′.mark==true then
 4 |   |   if N==N′ or N′! ∈ parent(N) then
 5 |   |   |   return one
 6 |   |   end
 7 |   end
 8 |   if cycleSearch(N′)==one then
 9 |   |   return one
10 |   end
11 end
```

The pseudocode for the cycle detection is shown in Algorithm 9.

6.2 Provide clear and sufficient reasons to show that the running time of your algorithm can be O(m + n) for a graph with n nodes and m edges.

The proposed algorithm is based on the Depth First Search (DFS). The time complexity of DFS is O(m+n) for a graph that contains n nodes and m edges. Reason is that:

Each node is visited once: n_1 – (edges on n_1) – n2 – (edges on n_2) – ..... – n_n – (edges on v_n).

This equals: (v_1+v_2+v_3+.....v_n) + (edges on v_1 + edges on v_2 + ... edges on v_n).

Because (v_1+v_2+v_3+.....v_n) = n, and (edges on v_1 + edges on v_2 + ... edges on v_n) = m, the total time complexity = m + n.

Therefore, the cost is the number of nodes plus the number of edges, which is O(m + n).

6.3 A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any binary tree the number of nodes with two children is exactly one less than the number of leaves.

Theorem: In any binary tree, if the number of nodes with two children is n, the number of leaves is $n + 1$.

Proof: By induction on a binary tree B(n), which has n nodes with two children, and n+1 leaves.

Base case: When n = 1, #leaves = 2. Thus, the number of nodes with two children is one less than the number of leaves. In other words:
the number of nodes with two children = n;
the number of leaves = (n + 1).

Induction hypothesis: B(n) is true, if the number of nodes with two children is n, the number of leaves is n+1.

Induction step: In the case of B(n+1), there exist two situations:

Firstly, a leaf is altered to a node, and this new node will have two leaves. In this case of B(n+1), we need to add one more node with two children; also, we need to deduct 1 leaf (that became a node with children) from the number of leaves, and add 2 new leaves. Hence, node number = n+1, leaves number = (n+1)-1+2 = n+2. Thus, the number of leaves is still one larger than the number of nodes with two children in the case n+1.

Secondly, a leaf is alternated to a node, and this new node only has one leaf. In this case, the number of two-children nodes is still n. We need to deduct 1 from leaves, and plus 1 for the new leaf, which means the number of leaves remains the same: (n+1)-1+1 = n+1. Thus, the number of leaves is one larger than the number of nodes with two children in the case n+1.

Corollary: In any binary tree, the number of two-children nodes is exactly one less than the number of leaves.