Knapsack Problem
Comparison Report

Greedy vs. Dynamic vs. Exhaustive

Samy Masadi
CSCI 423
April 12, 2019

Contents

1 Summary

The program knapsack_problem.py demonstrates and measures the performance of three different solutions to the 0-1 knapsack problem. The greedy ratio algorithm selects the best value/weight ratio item one at a time until the knapsack is filled. The dynamic programming algorithm determines the best value for a capacity and items available by building on previously calculated values that it has stored in memory. The exhaustive search is a brute force algorithm that iterates through every possible combination of items and keeps track of which both fits in the knapsack and has the highest value.

Both dynamic programming and exhaustive search produce the most optimal value combination of items, while the greedy ratio will not always be optimal. The exhaustive search is always optimal because it checks every combination. Dynamic programming considers whether each item will produce an optimal combination by comparing the value with its inclusion to the previously saved best value without its inclusion. The greedy method, meanwhile, is only concerned with the best ratio item at the time. Since it does not consider the whole combination, it will sometimes ignore items simply because they do not have the best ratio at the time of selection.

As far as time and space efficiency, the greedy and dynamic algorithms are much better than the exhaustive search. With any n and capacity beyond small values, brute force becomes prohibitively expensive. Since dynamic programming always produces an optimal result, it is the recommended algorithm for most circumstances. The only circumstances where one might want to choose the greedy algorithm is in memory-limited scenarios involving large numbers of items and capacities. The one advantage the greedy algorithm has is its space efficiency that scales with number of items or capacity. For this reason, one may choose it as a "good enough" option only if constrained by memory.

2 Methodology

2.1 Greedy Ratio Pseudocode

```
greedyRatio(items, capacity):
    while selecting items:
        for item in items:
            check if item was already selected
            check if item fits
            find/select best value/weight item
        if an item was selected:
            add to items selected
        else stop selecting items
    return items selected
```

2.2 Dynamic Programming Pseudocode

```
dynamic(items, capacity):
    init 2D graph
    populate graph's first row and column with 0s
    for row in graph:
        for col in graph:
            if col - item weight >= 0:
                graph[row].append(max(graph[row-1][col], item value + graph[row-1][col-item weight]))
            else graph[row].append(graph[row-1][col])
    backtrack to determine selected items
    return selected items
```

2.3 Exhaustive Search Pseudocode

```
bruteForce(items, capacity):
    for each combination size:
        for combo in combinations:
            if combo weight fits:
                determine if best value
    return best value combo
```

2.4 Testing

Each algorithm was tested on items and knapsacks of various sizes. Besides the example provided in the project requirements, two other item lists of size 20 and size 27 were created to show the practical limits of the exhaustive search algorithm. Two additional item lists of size 5000 and 10000 were created to show large item and capacity scaling for the greedy and dynamic algorithms. Also, to fully show the scaling of the algorithms, knapsack capacity was scaled equally along with number of items.

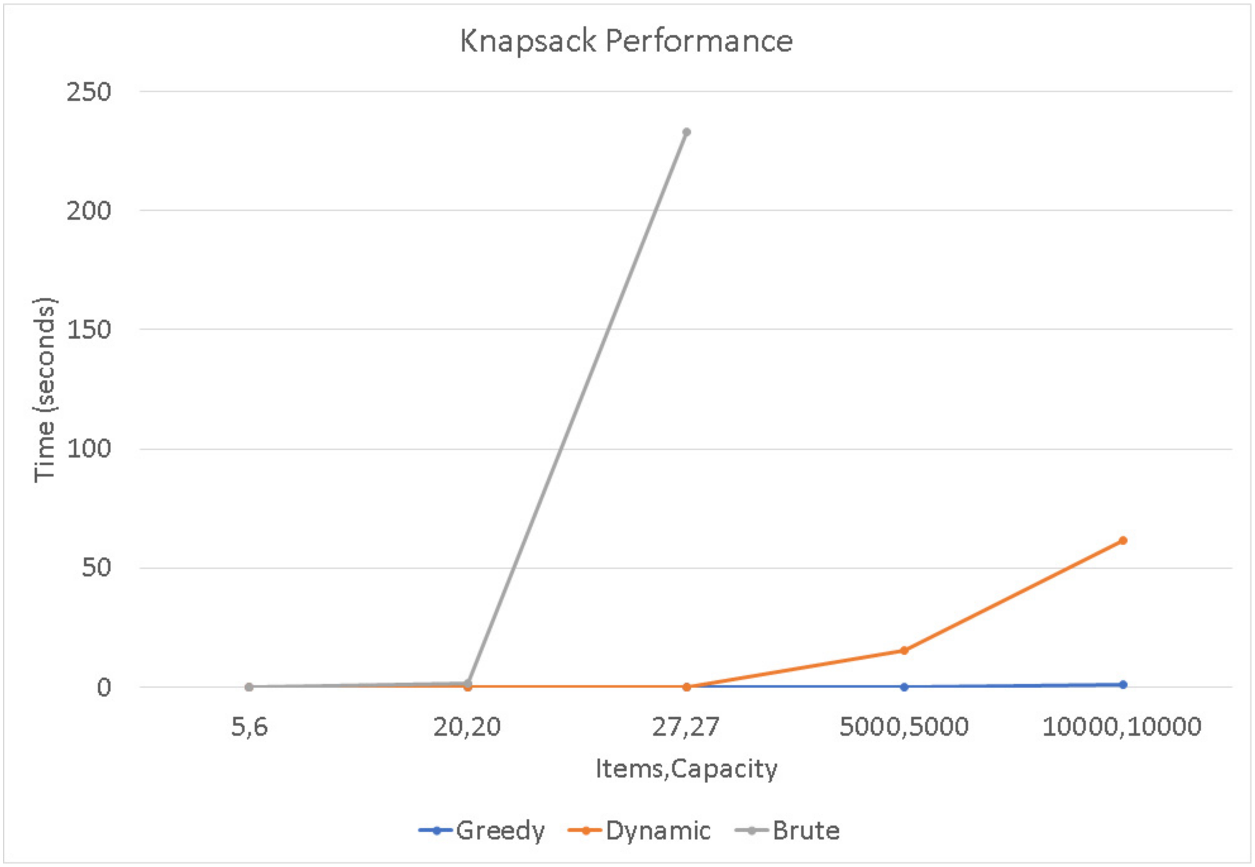
The item list for Test 2 (20,20) was constructed specifically to guarantee a demo in which the greedy ratio algorithm will not produce an optimal item selection. Since it always selects the best value/weight ratio first, it will ignore solutions that produce the best value, but do not have the best ratio. In Test 2, the greedy method with take the best ratio item first, but then will be forced to take suboptimal items for the remainder of the knapsack. The optimal selection is a single item that takes up the entire knapsack capacity by itself, which will be ignored because the method will have already selected another item.

3 Performance Reports

3.1 Size to Performance

Items,Cap	Greedy	Dynamic	Exhaustive
5,6	0.000016	0.000028	0.000033
20,20	0.000071	0.000246	1.416452
27,27	0.000054	0.000444	233.089384
5000,5000	0.242298	15.405498	N/A
10000,10000	1.030053	61.720472	N/A

3.2 Comparison Graph



4 Complexity Analysis

4.1 Efficiency per Algorithm

Algorithm	Time	Space
Greedy	O(n*c), O(n^2)	O(n), O(c)
Dynamic	O(n*c)	O(n*c)
Exhaustive	Θ(n!*n)	Θ(n!*n)

4.2 Time Efficiency

4.2.1 Greedy Ratio: O(n*c) or O(n^2)

The method needs to iterate through all items (n) to find the one with the best value-to-weight ratio. It keeps looping through this until either the knapsack is full or all items have been added. There is no certainty whether the total weight of all items (n) or the capacity (c) will be the limiting factor for the second dimension. It will be limited by whichever is smaller.

In reality, a situation where the total weight of all items is greater than the knapsack capacity is arguably more likely. So real-world performance scenarios will probably reflect O(n*c).

4.2.2 Dynamic Programming: O(n*c)

Since the method inherently fills up and utilizes a 2D list, the efficiency will always be limited by the list's dimensions of items (n) by capacity (c).

4.2.3 Exhaustive Search: Θ(n!*n)

The brute force method iterates through every possible combination of items from 1 to either number of items (n) or capacity (c). In the worst case, the greatest number of combinations always occurs at nC$\frac{n}{2}$, which scales with n!.

Then for each combination, the algorithm must also iterate through every item within and tally the weight and value. It calculates the combo's total weight to see if it fits in the knapsack, as well as determines if it represents the highest total value. In the worst case of nC$\frac{n}{2}$, the largest combination size thus equals $\frac{n}{2}$, which scales with n.

A best case scenario will occur as long as capacity (c) is less than $\frac{n}{2}$. In this case, efficiency will be Ω(nCc * c).

4.3 Space Efficiency

4.3.1 Greedy Ratio: O(n) or O(c)

The algorithms stores items selected in a list, and refers to the list when selecting subsequent items to avoid redundant selections. Thus, storage scales with either the total weight of all items (n) or capacity (c), whichever is smaller. In real-world scenarios, capacity will likely be the limiting factor.

4.3.2 Dynamic Programming: O(n*c)

At its core, the method depends on storing previously calculated values in a 2D list, so space efficiency will be the list's dimensions of items (n) by capacity (c).

4.3.3 Exhaustive Search: Θ(n!*n)

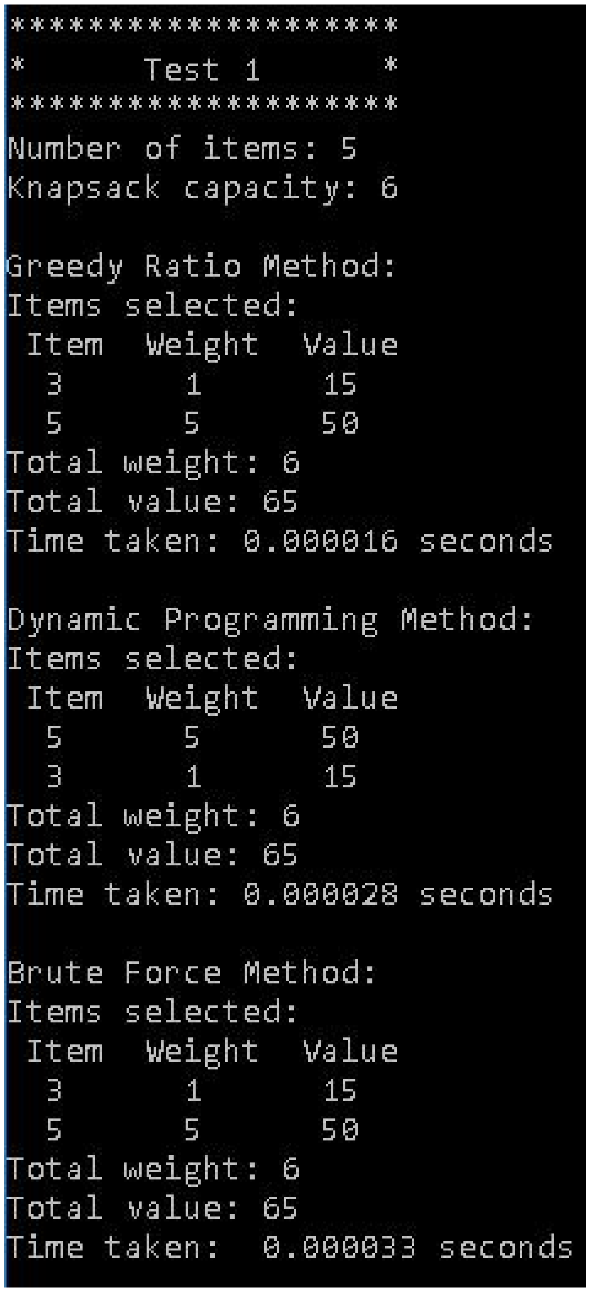
Since the method iterates through a generated list of combinations, the number of elements stored in memory will peak at nC$\frac{n}{2}$ * $\frac{n}{2}$, which essentially scales with n!*n.

As with time efficiency, the best case will occur when c is less than $\frac{n}{2}$. In this case, space efficiency will be Ω(nCc * c).

5 Schedule

Date	Plan	Completed
4/5	begin work on program	
4/7	complete exhaustive search	greedy ratio method
4/9	complete dynamic programming method	
4/10	complete third method	exhaustive search method
4/11	conduct time trials; begin report	dynamic programming method
4/12	complete program and report	program, time trials
4/13		report

6 Screenshot Showcase



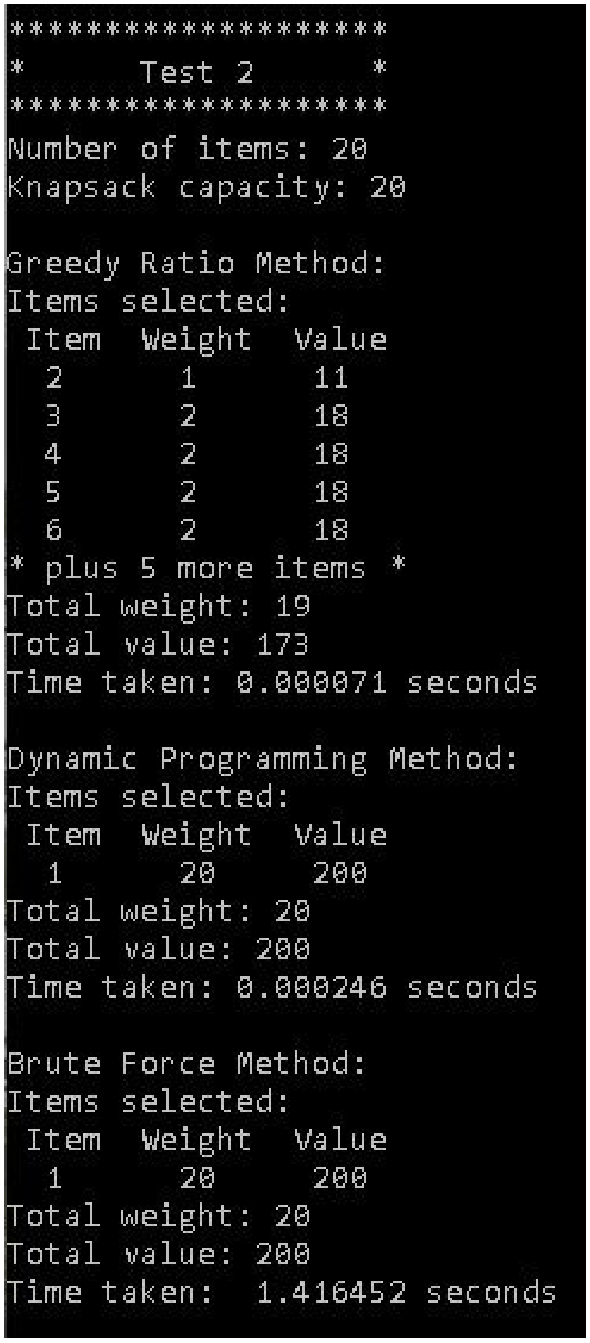
```
********************
*      Test 1      *
********************
Number of items: 5
Knapsack capacity: 6

Greedy Ratio Method:
Items selected:
 Item  Weight  Value
  3      1      15
  5      5      50
Total weight: 6
Total value: 65
Time taken: 0.000016 seconds

Dynamic Programming Method:
Items selected:
 Item  Weight  Value
  5      5      50
  3      1      15
Total weight: 6
Total value: 65
Time taken: 0.000028 seconds

Brute Force Method:
Items selected:
 Item  Weight  Value
  3      1      15
  5      5      50
Total weight: 6
Total value: 65
Time taken:  0.000033 seconds
```

Figure 1: Results for 5 items, 6 capacity.



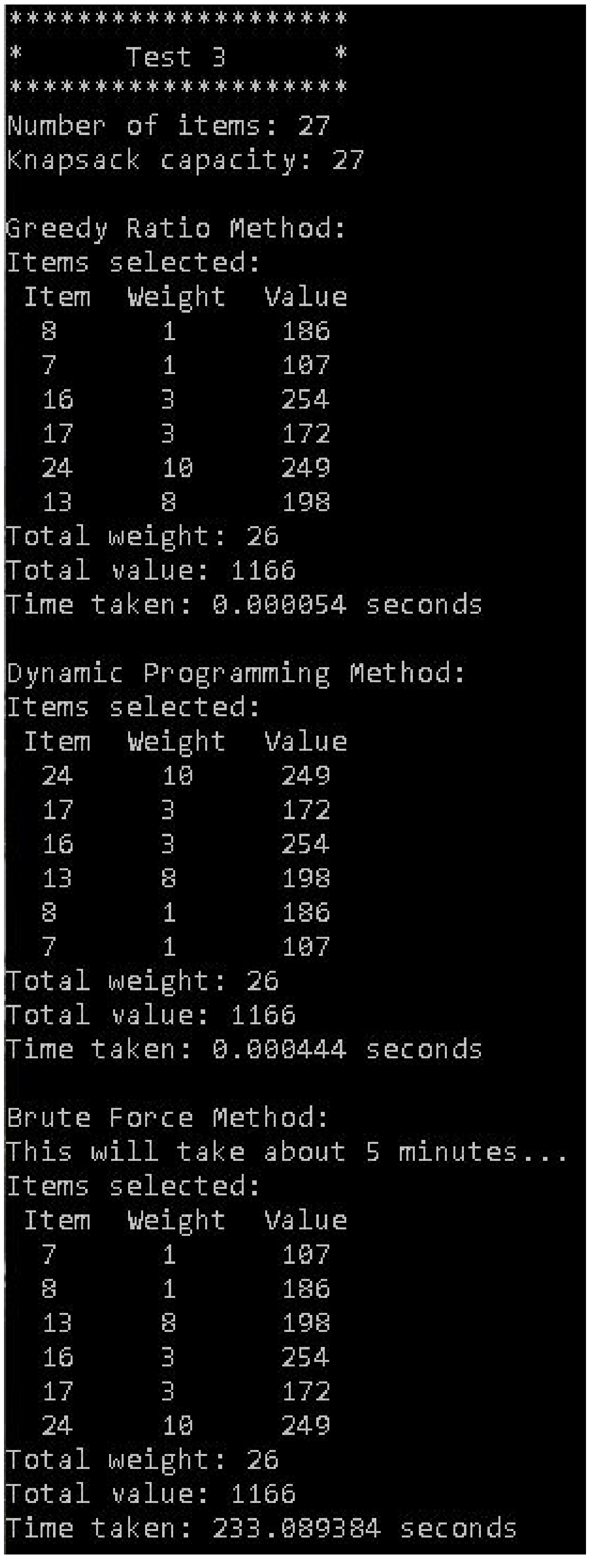
```
********************
*      Test 2      *
********************
Number of items: 20
Knapsack capacity: 20

Greedy Ratio Method:
Items selected:
 Item  Weight  Value
  2      1      11
  3      2      18
  4      2      18
  5      2      18
  6      2      18
* plus 5 more items *
Total weight: 19
Total value: 173
Time taken: 0.000071 seconds

Dynamic Programming Method:
Items selected:
 Item  Weight  Value
  1      20     200
Total weight: 20
Total value: 200
Time taken: 0.000246 seconds

Brute Force Method:
Items selected:
 Item  Weight  Value
  1      20     200
Total weight: 20
Total value: 200
Time taken:  1.416452 seconds
```

Figure 2: Results for 20 items, 20 capacity.



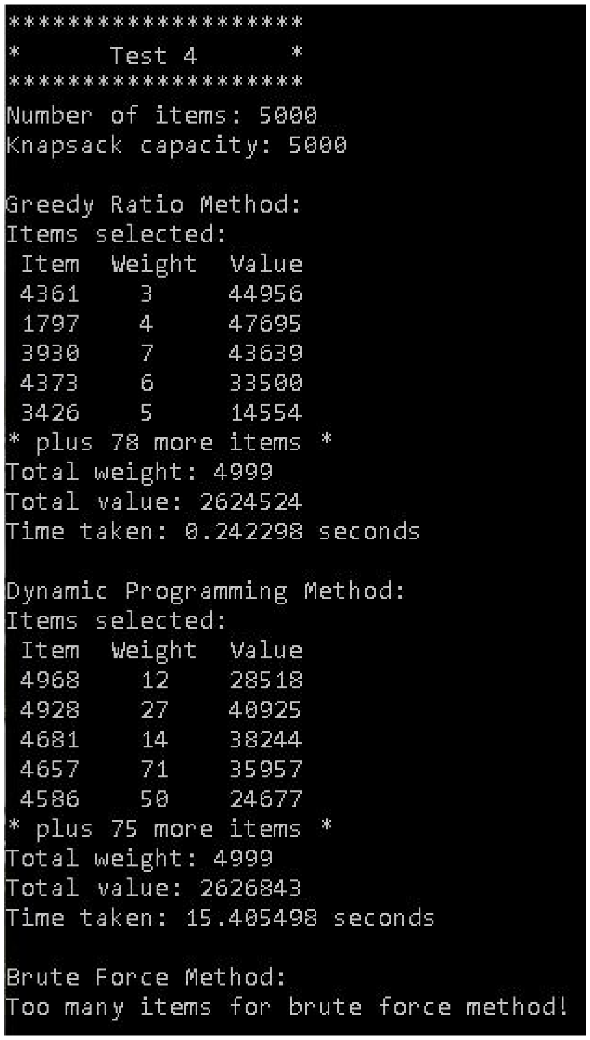
```
********************
*      Test 3      *
********************
Number of items: 27
Knapsack capacity: 27

Greedy Ratio Method:
Items selected:
 Item  Weight  Value
  8      1      186
  7      1      107
  16     3      254
  17     3      172
  24     10     249
  13     8      198
Total weight: 26
Total value: 1166
Time taken: 0.000054 seconds

Dynamic Programming Method:
Items selected:
 Item  Weight  Value
  24     10     249
  17     3      172
  16     3      254
  13     8      198
  8      1      186
  7      1      107
Total weight: 26
Total value: 1166
Time taken: 0.000444 seconds

Brute Force Method:
This will take about 5 minutes...
Items selected:
 Item  Weight  Value
  7      1      107
  8      1      186
  13     8      198
  16     3      254
  17     3      172
  24     10     249
Total weight: 26
Total value: 1166
Time taken: 233.089384 seconds
```

Figure 3: Results for 27 items, 27 capacity.



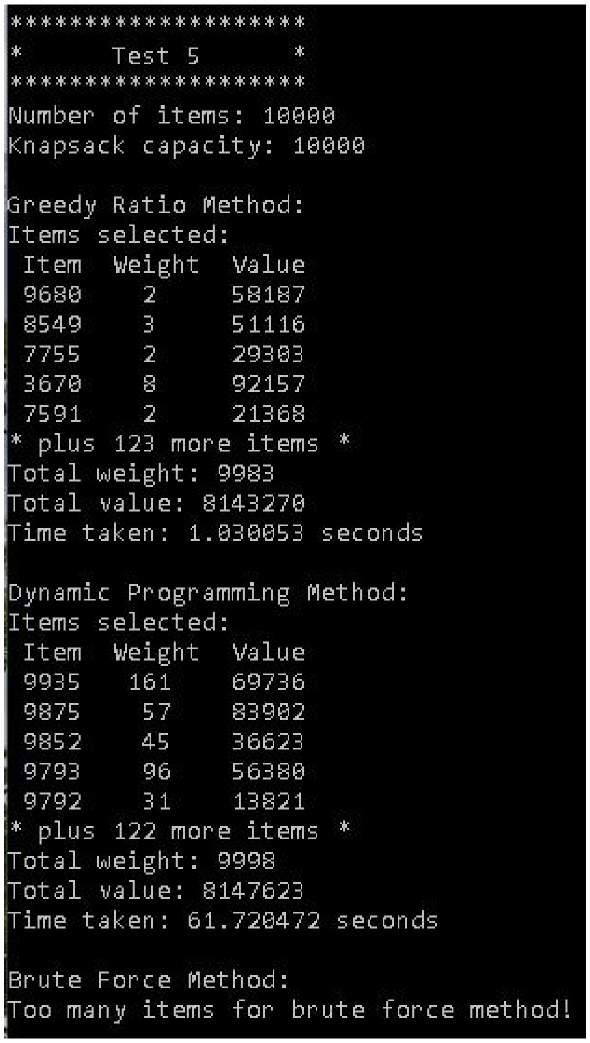
```
********************
*      Test 4      *
********************
Number of items: 5000
Knapsack capacity: 5000

Greedy Ratio Method:
Items selected:
 Item  Weight  Value
 4361    3     44956
 1797    4     47695
 3930    7     43639
 4373    6     33500
 3426    5     14554
* plus 78 more items *
Total weight: 4999
Total value: 2624524
Time taken: 0.242298 seconds

Dynamic Programming Method:
Items selected:
 Item  Weight  Value
 4968    12    28518
 4928    27    40925
 4681    14    38244
 4657    71    35957
 4586    50    24677
* plus 75 more items *
Total weight: 4999
Total value: 2626843
Time taken: 15.405498 seconds

Brute Force Method:
Too many items for brute force method!
```

Figure 4: Results for 5000 items, 5000 capacity.



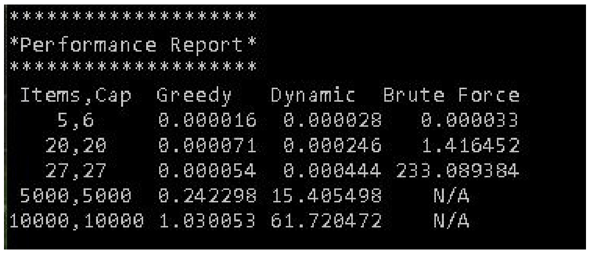
```
********************
*      Test 5      *
********************
Number of items: 10000
Knapsack capacity: 10000

Greedy Ratio Method:
Items selected:
 Item  Weight  Value
 9680    2     58187
 8549    3     51116
 7755    2     29303
 3670    8     92157
 7591    2     21368
* plus 123 more items *
Total weight: 9983
Total value: 8143270
Time taken: 1.030053 seconds

Dynamic Programming Method:
Items selected:
 Item  Weight  Value
 9935   161    69736
 9875    57    83902
 9852    45    36623
 9793    96    56380
 9792    31    13821
* plus 122 more items *
Total weight: 9998
Total value: 8147623
Time taken: 61.720472 seconds

Brute Force Method:
Too many items for brute force method!
```

Figure 5: Results for 10000 items, 10000 capacity.



```
********************
*Performance Report*
********************
 Items,Cap   Greedy    Dynamic  Brute Force
    5,6      0.000016  0.000028   0.000033
   20,20     0.000071  0.000246   1.416452
   27,27     0.000054  0.000444 233.089384
 5000,5000   0.242298 15.405498    N/A
10000,10000  1.030053 61.720472    N/A
```

Figure 6: Performance Report