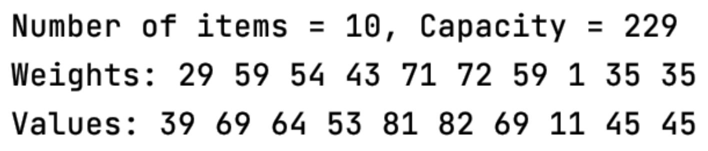
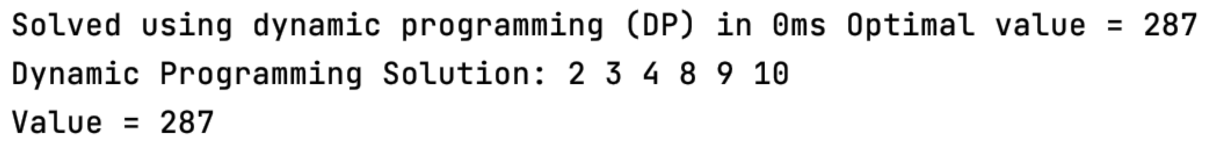
Melvin Evans

CSC 140 Advanced Algorithms
Assignment #4 Lab Report

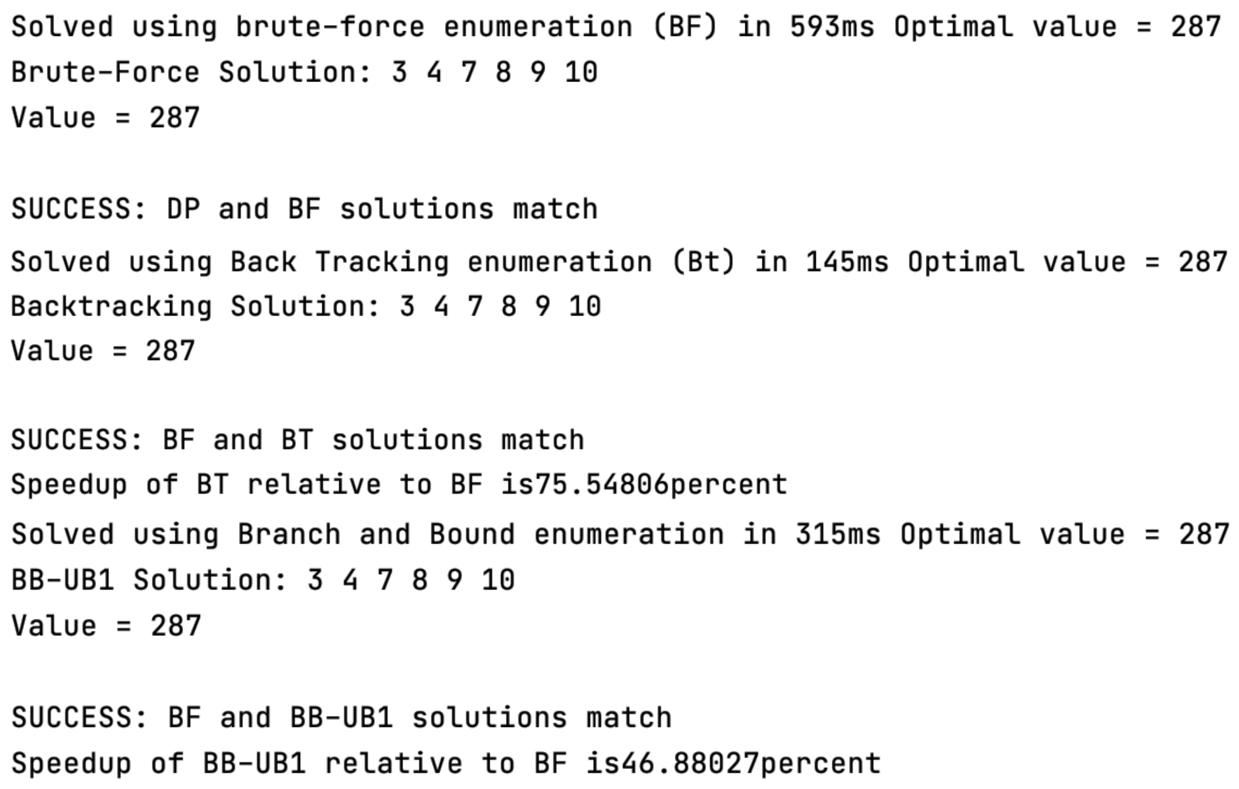
Proof of matching proper outputs



```
Number of items = 10, Capacity = 229
Weights: 29 59 54 43 71 72 59 1 35 35
Values: 39 69 64 53 81 82 69 11 45 45
```



```
Solved using dynamic programming (DP) in 0ms Optimal value = 287
Dynamic Programming Solution: 2 3 4 8 9 10
Value = 287
```



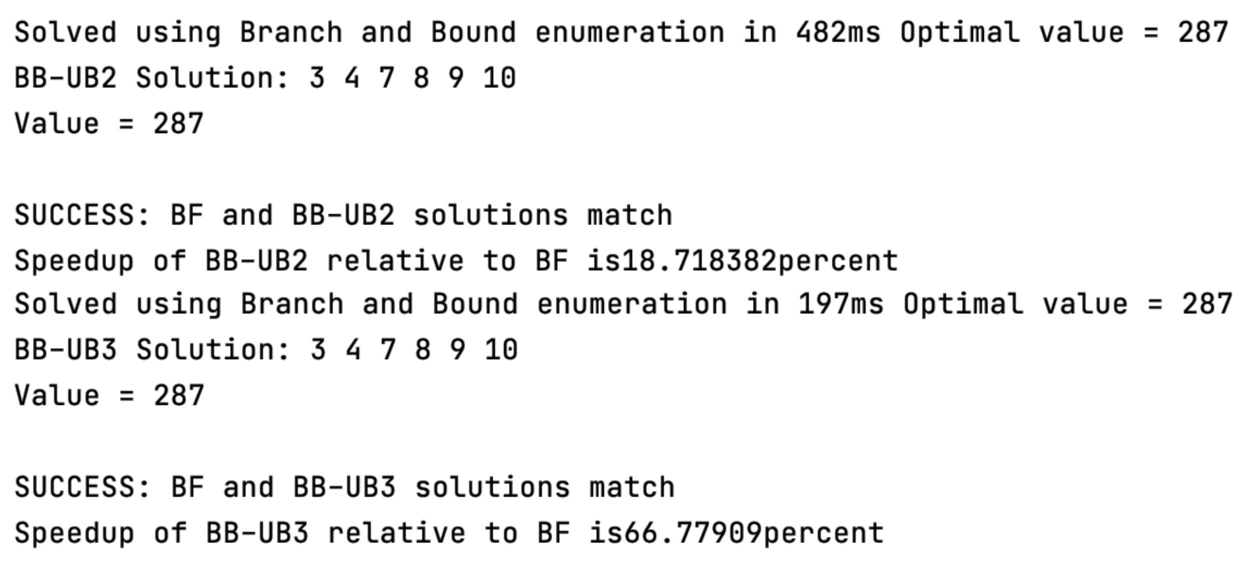
```
Solved using brute-force enumeration (BF) in 593ms Optimal value = 287
Brute-Force Solution: 3 4 7 8 9 10
Value = 287

SUCCESS: DP and BF solutions match
Solved using Back Tracking enumeration (Bt) in 145ms Optimal value = 287
Backtracking Solution: 3 4 7 8 9 10
Value = 287

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is75.54806percent
Solved using Branch and Bound enumeration in 315ms Optimal value = 287
BB-UB1 Solution: 3 4 7 8 9 10
Value = 287

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is46.88027percent
```

Melvin Evans



```
Solved using Branch and Bound enumeration in 482ms Optimal value = 287
BB-UB2 Solution: 3 4 7 8 9 10
Value = 287

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is18.718382percent
Solved using Branch and Bound enumeration in 197ms Optimal value = 287
BB-UB3 Solution: 3 4 7 8 9 10
Value = 287

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is66.77909percent
```

Table for Don't take first

	Brute Force	Backtracking	B&B UB1	B&B UB2	B&B UB3	Dynamic Programming
N = 30	91148ms	53986ms	41120ms	95429ms	113280ms	7ms

Search Order Discussion

The Take first option runs faster in comparison to the Don't take first option mainly due to how the order of the code execution. As in the don't take first option you start by building the tree with not taking any items and then slowing taking items in different combinations until you take all items. While the take items first approach builds the tree from left to right first taking all items possible and then running combinations with less items taken until it takes zero items. The reason why the take items first option is expected to be faster is that the goal of the 0-1 knapsack problem is to fill the knapsack all the way as much as possible and find the maximum value. Therefore, starting on the taken item side first with methods that return early like backtracking and Branch and Bound allow the solution to be found faster. As you do not have to get through all of the cases where the knapsack is not filled nearly to its capacity. While at the same time will avoid overfilling the knapsack due to the backtracking. The only time this won't lead to a speed up is when looking at the simple upper bound or upper bound 1 as this one is based off of the number of untaken values in a given tree. Therefore, it would be faster to start by looking at the values with less items taken in it.

Table for Take first

	Brute Force	Backtracking	B&B UB1	B&B UB2	B&B UB3	Dynamic Programming
N = 10	2ms	2ms	0ms	0ms	2ms	1ms

Melvin Evans

N = 20	121ms	70ms	81ms	130ms	221ms	1ms
N = 30	81598ms	49792ms	55084ms	55234ms	73365ms	6ms
N = 40	Timed out	Timed out	Timed out	Timed out	Timed out	11ms
Largest Input Solved in 10s	26	27	27	27	27	Unknown ran out of memory before mark was found to be able to test size

For all the different running conditions of the knapsack problem they all got exponentially bigger as the input got larger and larger. With all of them capping out when they were ran given the input size of 40 except for Dynamic programming. Which survived that sequence due to the fact that we are just populating a table and deciding one route versus building a whole tree to hold each sub problem in. Overall, the fastest besides dynamic programming seemed to be the Backtracking which was a bit weird as in the B&B problems we were also doing backtracking just with an extra conditional that should have made it on paper end faster than just doing backtracking by itself.

In terms of the largest input size for each of the different running conditions they all were around the same except with the notable cases of brute force which was 26 instead of 27 and dynamic programming which couldn't be tested with the given code to confirm due to lack of memory. For the brute force this difference is due to the fact that brute force has to check all of the nodes of the tree no matter if it has already found the best solution of the knapsack. While dynamic programming was by far the fastest due to the nature of it therefore it would require a size bigger than we could test in this environment in order to properly quantify the amount needed to reach 10 seconds. Lastly all of the rest of them were found to undershoot 10 seconds at an input size of 27 but when they were given a size of 28, they all over shot 10 seconds. Therefore 27 is the max input size that those configurations could handle to solve the knapsack problem.