Search Order

Note that in the given templates, the Don't-take option is explored before the Take option. This is not the best search order. Exploring the Take option first is expected to be a better search order that completes the search faster, why?

When the "Take" option is the first one, the algorithm searches through paths that accumulate value faster and since our goal is to maximize the total value within the weight capacity, we add items earlier. The algorithm quickly builds up a solution with potentially high total value, which could be close to the optimal solution. This approach allows us to prune all the branches with a value less than our best.

In our branch-and-bound algorithms, the bounding step uses the current best solution to determine if a particular branch can potentially lead to a better solution or not. By exploring Take first, the algorithm quickly finds a larger solution by increasing the bound and allowing us to prune other branches much more aggressively.

Discussion

The table below shows all the different timings for different input sizes, as input size grows, the timing grows exponentially since we are dealing with O(2^n) time complexity.

Brute Force: The brute force approach shows an exponential increase in time as the size of N grows.

Backtracking: The backtracking approach performs better than brute force. It provides some improvement by pruning the trees, but times out at N = 40.

Branch and Bound (B&B) with Upper Bound 1 (UB1): This strategy offers a better speedup compared to backtracking, particularly notable at N = 30 but still results in a timeout at N = 40. The improvement gives more effective pruning but still suffers from exponential complexity.

B&B with Upper Bound 2 (UB2): There is a significant speedup at N=30, and it can solve N=40 in about 13 minutes. Therefore, the tighter the upper bound the more chance to prune it.

B&B with Upper Bound 3 (UB3): This strategy improves even further, solving N=30 instantly and handling N=40 just as quickly as dynamic programming. The upper bound is chosen much more aggressively which helps us prune much earlier.

Dynamic Programming: Solves all provided input sizes instantly, due to its pseudo polynomial time complexity. The only downside is that it uses a lot of memory compared to all the other ones.

	Brute Force	Backtracking	B&B UB1	B&B UB2	B&B UB3	Dynamic Programming
N = 10	0 ms	0 ms	0 ms	0 ms	0 ms	0 ms
N = 20	46 ms	27 ms Speedup = 41.30%	16 ms Speedup = 65.22%	2 ms Speedup = 95.65%	0 ms Speedup = 100%	0 ms Speedup = 100%
N = 30 **Only For Don't Take First**	48584 ms	29561 ms Speedup = 39.15%	17729 ms Speedup = 63.15%	810 ms Speedup =98.33%	0 ms Speedup = 100%	0 ms Speedup = 100%
N = 40	Timeout	Timeout	Timeout	829,440 ms	1 ms	1 ms
Largest Input Solved in 10s	24 ~ 25	27	27~28	32~33	50~51	The max that I managed to get on the ECS machine without causing Core Dump was 288. With 3 ms

Note: Since some algorithms take 0 ms, the speedup shows as 0, but it's intentionally implemented in the given template.

Results of input size 10 with Dont -Take going first

```
[igordyy@ecs-pa-coding1 Assignment4]$ ./a.out 10
Number of items = 10, Capacity = 323
Weights: 84 87 78 16 94 36 87 93 50 22
Values: 94 97 88 26 104 46 97 103 60 32


Solved using dynamic programming (DP) in 0 ms. Optimal value = 383
Dynamic Programming Solution: 1 2 3 4 6 10
Value = 383


Solved using brute-force enumeration (BF) in 0 ms. Optimal value = 383
Brute-Force Solution: 1 3 4 6 7 10
Value = 383

SUCCESS: DP and BF solutions match

Solved using backtracking (BT) in 0 ms. Optimal value = 383
Backtracking Solution: 1 3 4 6 7 10
Value = 383

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is 0.00%

Solved using branch-and-bound (BB) with UB1 in 0 ms. Optimal value = 383
BB-UB1 Solution: 1 3 4 6 7 10
Value = 383

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is 0.00%

Solved using branch-and-bound (BB) with UB2 in 0 ms. Optimal value = 383
BB-UB2 Solution: 1 3 4 6 7 10
Value = 383

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is 0.00%


Solved using branch-and-bound (BB) with UB3 in 0 ms. Optimal value = 383
BB-UB3 Solution: 1 3 4 6 7 10
Value = 383

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is 0.00%

Program Completed Successfully
```

Result of input size 10 with Take going first

```
[igordyy@ecs-pa-coding1 Assignment4]$ ./a.out 10
Number of items = 10, Capacity = 323
Weights: 84 87 78 16 94 36 87 93 50 22
Values: 94 97 88 26 104 46 97 103 60 32


Solved using dynamic programming (DP) in 0 ms. Optimal value = 383
Dynamic Programming Solution: 1 2 3 4 6 10
Value = 383


Solved using brute-force enumeration (BF) in 0 ms. Optimal value = 383
Brute-Force Solution: 1 2 3 4 6 10
Value = 383

SUCCESS: DP and BF solutions match

Solved using backtracking (BT) in 0 ms. Optimal value = 383
Backtracking Solution: 1 2 3 4 6 10
Value = 383

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is 0.00%

Solved using branch-and-bound (BB) with UB1 in 0 ms. Optimal value = 383
BB-UB1 Solution: 1 2 3 4 6 10
Value = 383

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is 0.00%

Solved using branch-and-bound (BB) with UB2 in 0 ms. Optimal value = 383
BB-UB2 Solution: 1 2 3 4 6 10
Value = 383

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is 0.00%


Solved using branch-and-bound (BB) with UB3 in 0 ms. Optimal value = 383
BB-UB3 Solution: 1 2 3 4 6 10
Value = 383

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is 0.00%

Program Completed Successfully
```