Hunter Brown

CSC140 Advanced Algorithms
CSC206 Algorithms and Paradigms
Fourth Assignment: Branch-and-Bound, Dynamic Programming and Greedy Algorithms

Exploring the Take option first is expected to be a better search order that completes the search faster, why?

If you do the Take option first before the Don't-take option, you are more likely to arrive at a larger solution quicker, thus making the upper bound much tighter and closer to the best possible solution sooner and allowing you to do more pruning compared to if you did don't-take every time, which would have to go all the way down to the last item potentially before it can form an upper bound (i.e. nonzero) and begin to use it for pruning. Having more branch prunes also means the algorithm in general will need to do fewer recursive calls, reducing the overhead and overall reducing the computation time as many steps are now cut out.

DATA TABLE

	Brute Force	Backtracking	B&B UB1	B&B UB2	B&B UB3	Dynamic Programming
N = 10	426ms	169ms	280ms	7ms	2ms	1ms
N = 20	620.4s	271.9s	33.5s	19ms	14ms	2ms
N = 30	TIME OUT	TIME OUT	383.9s	1.321s	27ms	3ms
N = 40	TIME OUT	TIME OUT	TIME OUT	127.2s	28ms	4ms
Largest Input Solved in 10s	n=14	n=15	n=15	n=35	10000	10000

Note: 10,000 is the max I can really go before I start getting stack overflow errors or run out of heap space; however, even at 10,000 it only takes about 1 second for UB3, indicating this can solve even larger input sizes in under 10 seconds.

SCREENSHOTS

DP.)

```
Solved using brute-force enumeration (BF) in 426ms Optimal value = 333
Brute-Force Solution: 4 5 6 7 8 9 10
Value = 333

SUCCESS: DP and BF solutions match
```

BT.)

```
Solved using Back Tracking enumeration (Bt) in 169ms Optimal value = 333
Backtracking Solution: 4 5 6 7 8 9 10
Value = 333

SUCCESS: BF and BT solutions match
Speedup of BT relative to BF is60.32864percent
```

BB UB1.)

```
Solved using Branch and Bound enumeration in 47ms Optimal value = 333
BB-UB1 Solution: 4 5 6 7 8 9 10
Value = 333

SUCCESS: BF and BB-UB1 solutions match
Speedup of BB-UB1 relative to BF is88.96713percent
```

BB UB2.)

```
Solved using Branch and Bound enumeration in 7ms Optimal value = 333
BB-UB2 Solution: 4 5 6 7 8 9 10
Value = 333

SUCCESS: BF and BB-UB2 solutions match
Speedup of BB-UB2 relative to BF is98.356804percent
```

BB UB3.)

```
Solved using Branch and Bound enumeration in 2ms Optimal value = 333
BB-UB3 Solution: 4 5 6 7 8 9 10
Value = 333

SUCCESS: BF and BB-UB3 solutions match
Speedup of BB-UB3 relative to BF is99.53052percent

Program Completed Successfully
```

DISCUSSION

BRUTE-FORCE ALGORITHM

To begin, let us discuss the results from the brute force algorithm. It is unsurprising that the brute-force method is by far the slowest algorithm out of the rest, since it checks each and every single possible solution before finishing the algorithm, meaning that it is a guaranteed runtime of $\Theta(2^n)$ every single time, since for each and every node there is are 2 possible choices that can be made (take or don't take), meaning that the total number of combinations is 2 * 2 * … 2 (n # of 2's multiplied together) which is 2^n.

Looking at the values themselves in the table, we can see that brute-force starts off at around 426ms at n=10, then 620,400ms at n=20, and then for n=30 and n=40 it in fact went on for over an hour and thus timed out. Comparing the time for n=10 and n=20 however, we can see that it's an increase in the number of items by a factor of 2 (10 * 2 = 20) from the first run to the second. Using this and plugging it into the runtime estimate, we get 2^{20} = ~1,000,000, and 2^{10} = ~1,000, meaning the increase in the amount of time it takes to run the program goes up by 2^{20} / 2^{10} = 2^{10} = ~1000 times, which is indeed what we roughly see between 426ms and 620,400ms (specifically, 1456 times more in this case).

For the final row in the table, we see that this algorithm can only solve an input size of 14 in under 10 seconds, which is honestly quite horrible (and, given the asymptotic complexity, makes perfect sense and why we absolutely need to think of a better, smarter solution to reduce the runtime; hence, the following attempts at improving the algorithm).

BACKTRACKING ALGORITHM

Next, the backtracking algorithm has a very simply addition to the brute-force method, and yet it saves a substantial amount of time. All the backtracking algorithm has added to it is (as the name implies) a backtracking method, aka it keeps track of the load (weight so far out of all items taken) and passes it down as a parameter for the next recursive call of the function (that way, it's only an O(1) runtime for each node to calculate the total weight). Using this at every node, we can check to see if the current load plus the currently-being-decided item's weight will exceed the capacity, and if this is the case, you should not try to take the item as it will lead to an invalid solution (instead, simply return if you "take" after "don't take," or just go to "don't take" and not take at all if you have your algorithm "take" first before "don't take"). This, in comparison to brute force (which checks every possibility), is a great improvement already since it will no longer guaranteed check every single possible combination (however, a runtime of $O(2^n)$ is still possible in case, somehow, the weights of all items combined do not exceed the capacity, in this case it will run essentially identical to brute-force since it will never exceed the capacity).

Looking at the times in the table, we see a very similar spread compared to brute-force, but on average what appears to be 3 times faster. For n=10, BT has a 169ms runtime, for n=20 a 271.9s or 271,900ms runtime, and for n=30 and n=40 it also times out (over an hour runtime) similar to brute-force. Comparing the times for n=10 and n=20, 271,900/169 = 1608, which is almost the same as the factor brute-force increased from n=10 to n=20. Comparing backtracking to brute-force's times, we see that 426/169 = 2.52 and 620,400/271,900 = 2.28, with an average ratio of roughly 2.4. Using this, we can estimate the asymptotic complexity for the backtracking algorithm is about $\Theta(1/2.4 * 2^n)$, or simplified to $\Theta(2^n)$, which is the same as brute-force (however, counting the constants, backtracking is just over 2 times faster, but this does not make a dent in comparison to an exponential).

Like previously with the brute-force algorithm, this algorithm can only solve an input size of a mere 15 in under 10 seconds, only 1 more than brute-force (still, very horrible, which makes sense due to the $\Theta(2^n)$ runtime complexity).

Hunter Brown

BRANCH AND BOUND ALGORITHMS

After backtracking, we will now analyze the differences between the three different branch and bound algorithms: UB1 (uses a simple, loose upper bound), UB2 (a more tight upper bound but still quite loose), and UB3 (a very tight and useful upper bound).

UB1

The UB1 branch and bound algorithm is a decent upgrade to the previous brute-force and backtracking algorithms in terms of runtime; the fact that it could even run the n=30 algorithm in less than an hour (let alone, 10 minutes) is a noticeable improvement off the bat. The upgrade in this algorithm comes from an upper bound being used to prune potential branches; specifically, it calculates the difference between the total value of all items and the total value of untaken & decided items so far, which leaves you with the total value of items taken plus all the undecided items (and assuming you were to take all the rest you haven't decided on yet, this value will be the max value you can possibly get from that node onwards). This, however, is a very loose bound since it doesn't account for items that *would* even fit in the remaining capacity after all the taken items so far (a fix that will be implemented with UB2).

Looking at the table, we see that for n=10 we get a runtime of 47ms, n=20 we get 33,500ms, n=30 we get 383,900s, and n=40 we get a timeout since it took longer than an hour. Just at a quick glance, we can see that the runtime roughly increases by a factor of 10 each time (383,900/33,500 = 11.5). With a near 10x increase in the runtime from n=20 to n=30 and with the input size increasing by a factor of 3/2 or 1.5, to get such a 10x increase in the runtime from just a mere 1.5x increase to the input size it would need to follow an exponential asymptotic complexity of sorts, aka the same $\Theta(2^n)$ runtime we've seen with the previous 2 algorithms thus far.

The final row in the table further proves this is the case, since the brute-force algorithm could only handle an input size of 14 and backtracking with an input size of 15; here, UB1 can only handle an input size of n=15 in under 10 seconds, hinting that it indeed explodes in runtime duration with the smallest increases in input size exactly like the previous 2 algorithms.

UB2

Here, things take a change for the better with the UB2 branch and bound algorithm. This algorithm uses a tighter upper bound (although, a still loose one) in comparison to the UB1 algorithm; essentially, at every node we calculate the total value of taken items so far plus all the undecided items *that would fit* in the remaining capacity left over after the currently taken items at that point. This means that our upper bound will be either the same as our UB1 bound, or smaller (and this typically will be the case, since the more you go down the tree and the more you take, the less your remaining capacity will be meaning far fewer items you could fit into it, especially ones with larger weights). However, this upper bound is still quite loose and I will discuss soon why that leads to problems.

With this implementation though, we can actually run input sizes larger than 14 or 15 in under 10 seconds; in fact, we can run an input size up to 35 in under 10 seconds with this algorithm! An improvement, however, since it takes nearly 10 seconds just to solve an input size of 35, this is another major hint that we are still in the realm of the exponential (and due to its still-loose bound that leaves room for a lot of branches to be explored that wouldn't be otherwise if we had a tighter bound; this algorithm, as we will see more in-depth shortly as well, has the same possible worst runtime as the previous 2 algorithms since its loose bound could wind up being just as not-so-useful as UB1's).

Peering at the table's values for the UB2 algorithm, we see that it can solve an input size of 10 in 7ms, a size of 20 in 19ms, 30 in 1321ms, and 40 in 127,200ms. Dividing 127,000/1321 we get 96, or roughly a 100x increase in the runtime just from n=30 to n=40. For a 4/3x factor increase in the input size, we get a ~100x factor increase in the runtime. Unfortunately, as we've already guessed before, this indeed means that it is an exponential (2^n) algorithm; however, it is still noticeably better than the previous 3 algorithms, meaning we are on the right track to reducing its runtime down to something more reasonable.

The final row of the table further seals the deal, given that we can only solve an input size of a measly 35 in under 10 seconds just shows that it is a major cost in time for each new input we add to the list (increasing its size).

UB3

Lo and behold, we finally arrive at our best branch and bound algorithm out of the three discussed here. Taking a look at the table, we can see that for an input size of n=10 we get a runtime of 2ms, for n=20 a runtime of 14ms, n=30 we get 27ms, and n=40 we get 28ms! Dividing 28/27, we get roughly 1.037, a number very close to 1 (or in other words, no change in the runtime despite the increase in the input size)! As a matter of fact (and judging by the result in the final row), we can run a substantial number of items and still be able to solve it in a reasonable time!

For my implementation, I could run up to around 10,000 items in just a second before I began to get stack overflow errors (likely due to the crazy amount of recursive function calls). Taking 1000ms for n=10,000 and 28ms for n=40 and dividing the two (1000/28), we get roughly a 35x factor increase in the runtime duration from n=40 to n=10,000, whereas the factor increase from 40 to 10,000 is exactly 250. This means that the runtime increase factor is less than the increase factor in the input size, meaning this follows more of a (sub) linear asymptotic complexity instead of an exponential one! Specifically, (log(1000) - log(28))/(log(10,000) - log(40)) = a ~0.65 growth rate, giving us an asymptotic complexity / runtime estimate of $\Theta(n^{0.65})$, which is incredibly better than a runtime estimate of $\Theta(2^n)$ like we've gotten previously.

This tremendous increase in runtime performance is due to the tight upper bound we calculate at each and every node with the help of the fractional knapsack algorithm. Essentially, the fractional knapsack problem will provide us a solution that is the best possible total amount that the 0-1 knapsack problem will either match, or be worse than. If we tweak the fractional knapsack algorithm, we can make it so it doesn't add the final "slack" item (aka the item we

can't/don't take the full amount of, but a fraction of it instead). This result will act as a perfect lower bound for our algorithm, and we can check at each node to see if its upper bound is better than the best solution (or lower bound of the whole problem to start) found so far (and if it isn't, don't even bother checking down that branch because at most you will get something that's worse than the best solution you've already found). If we also order our items before we begin the solver in descending order based on their value-per-weight, we can calculate the lower/upper bound at each node in O(n) time instead of $O(n^2)$ time (a massive improvement). Due to this upper bound being even smaller than UB1's and UB2's, this will allow for even more pruning (getting rid of invalid or worse solutions in bulk) and an even more efficient algorithm.

DYNAMIC PROGRAMMING ALGORITHM

For the final algorithm, we have our dynamic programming solver for the 0-1 knapsack problem. This algorithm genuinely amazed me how fast it could solve the entire problem despite being a nested for loop inside of another.

Glancing at the table, we see that for an input size of n=10 we solve it in a swift 1ms, for n=20 a time of 2ms, for n=30 a time of 3ms, and n=40 with a time of 4ms. For an input size of 10,000 that I did, it could solve the problem in just about 1,000ms, or 1 second. Given a 1ms runtime for n = 10 and a 2ms runtime for n=20 (including 3ms for 30, 4ms for 40, ~1000 for 10000), this means that an increase in the input size from 10 to 20 (or 2x) increases the runtime only from 1ms to 2ms or *also* by a factor of 2x; this implies it has a runtime estimate of $\Theta(n)$, a linear asymptotic complexity and a massive improvement compared to the original algorithm we had with the brute-force approach!

This major improvement in efficiency is thanks to the dynamic programming nature of our algorithm; moreover, with dynamic programming in general (and here with this algorithm) we solve all the smaller cases and reuse previously solved smaller cases to build up more and more answers, eventually giving us all the possible solutions (including the best solution at the end) *without* having to recalculate any of the ones we've already calculated! This is a major improvement in comparison to the brute-force/BT/UB1/UB2 approaches since we don't have the possibility of having to recalculate/revisit a node potentially dozens or hundreds of times as we go through the tree; each node is calculated exactly one time and one time only. To be precise, our algorithm will explore n * m nodes (n beings the number of items, and m being the capacity of the knapsack; the size of its table being n x m); hence, a runtime of O(mn), not just simply O(n) as we assumed before. Since the capacity is typically larger than there are items in the list, this would mean that O(mn) would be (on average) a larger runtime than $O(n^2)$. However, it still appears to follow a more linear-like runtime complexity, perhaps other underlying optimizations may be at play having to do more with the hardware side in running the program.