Name: Kevin Cisneros-Cuevas
Email: kcisne04@calpoly.edu

Introduction:
This write up and assignment had us explore two different solutions to the problem known as 0-1 Knapsack. The two solutions that we explored were the Dynamic Programming solution and the Branch and Bound solution, and the goal was to compare the solutions and see what the pros and cons of each are.

Summary Table:

	Easy20	Easy50	Hard50	Easy200	Hard200
Dynamic Program	Execution time: .48 ms Weight: 519	Execution time: .6 ms Weight: 250	Execution time: 4.36 ms Weight: 10110	Execution time: 4.73 ms Weight: 2658	Execution time: 53.49 ms Weight: 112648
Branch and Bound	Execution time: 2.52 ms Weight: 519	Execution time: 11.17 ms Weight: 250	Execution time: 151.03 ms Weight: 10110	Execution time: 93.82 ms Weight: 2658	java.lang.OutOfMemoryError: Java heap space

Branch and Bound Hard 200 Error
This error occurred because it exceeded the heap space of my computer so it ran out of space when running the program. The solution to this that I can explore is better memory usage since I am using a lot of memory for my Priority Queue.

Dynamic Programming Approach:
A. Description of important data structures and key points of the algorithm
The data structure that I used in my Dynamic Programming approach is a 2 dimensional array or a matrix that has integer values in each cell containing the best possible value for that cell. The rows in the matrix represent the current number of items that we are taking into account and the columns represent the capacities that we are testing all the way up until the capacity given. This way, we ensure that the maximum value will be held in the bottom right corner of the matrix once we are done filling in the table.

B. Time complexity of your implementation – theoretical and empirical
The theoretical time complexity of this implementation is of O(nW) where n represents the number of items and W represents the maximum capacity that we are given. The reason for this is because, when filling in the table, we have to loop over the rows and columns which are represented by the number of items and the capacity value respectively. Based on my calculations the empirical formula turned out to be more like O(n) when we had smaller values but when there were more values, it started to behave more like O(n^2).

C. Pros and Cons

Pros:

- I think that one of the advantages of doing this implementation is the behavior that it gives even with larger input sizes. Since the behavior is O(nW) then it performs that way with bigger number of items and stays normally consistent
- Another advantage is the fact that this solution almost always guarantees to find the optimal solution.

Cons:

- I would say that one of the cons for this implementation is the memory usage, since we are storing everything in a 2D array, we are using up memory in order to get more time.

D. Possible improvements

I think one improvement that I can think of, that can use a bit less memory, is figuring out some way to use memoization, so that we do not make redundant calculations and use less space in the table that we make. This could also allow us to use a 1D array instead and if we used a top down approach iteratively we could reduce the time complexity to O(W) and reduce the space usage also to O(W).

Branch and Bound Approach:

A. Description of the search strategy and bounding function:

The search strategy that I used involves a Priority Queue implemented by the Java library. I loop through the items and make nodes that calculate the node if I were to take the item or leave the item out and then check if they exceed the capacity or are promising branches then I add them to my queue to explore the branch further. My bounding function consists of checking if we are at capacity or not. If we have not exceeded the capacity then we loop through each item and we keep adding its value to the bound as long as it does not exceed the capacity. And when we finish through the loop, if there are still items left then that means that the knapsack is not completely full and calculate the bound based on the potential values of adding the items to the sack based on their value to weight ratio.

B. Description of the data structures used and key points of the algorithm:

The data structures that I used for this implementation, were an Item class that represented an item that had a value, weight, item number identifier, and its value to weight ratio. The other data structure that I used was a KnapSackNode which was a node that I used to put in the Priority Queue for my algorithm. This KnapSackNode has information like, value, weight, level, index, bound, and the previous parent in order to backtrack when we find the best solution. Some of the key points of my algorithm are when I created the dummy node to start at the root of the tree and then iterating until the queue is empty and making children nodes. The children nodes were created based on whether we add the item to the next item or not and analyzed based on their bound and only added to the queue when their bound was higher than the current max value that I had.

C. Analyze and discuss both the theoretical and empirical time complexity:
The theoretical time complexity of this implementation comes out to be O(2^n) which is exponential and can be seen when run. Empirically based on my calculations and the times that I got, I would say that it is no better than O(n^2) with easier problems, but with harder problems it really does begin to become more like O(2^n).

D. Pros and Cons
Pros:

- One of the advantages of this implementation is that it prunes the nodes that are not promising and saves some time
- This implementation also helps with other sort of problems that are not easily solved with other implementations like fractional knapsack and multiple knapsacks

Cons:

- One of the cons for this implementation is seen in its time complexity. Although the worst case is O(2^n) and there is factors you can change to mitigate that, like better hardware and such, there is still the case where it is O(2^n) and that is not something that is favorable for this implementation
- Another disadvantage that this has is the fact that it uses up a lot more memory than some other solutions since I am using a Priority Queue that takes up more memory the more items we have to consider.

E. Possible improvements:
One of the possible improvements that I can come up with is possibly adding some of the techniques from Dynamic Programming like memoization or having a table to reduce repetitive computations that can help with some of the memory but also some of the time.

Conclusion:
In conclusion, the 0-1 Knapsack problem taught me the different ways that one problem can be solved and the tradeoffs that come with using a different approach. Dynamic Programming showed consistency and better time, but Branch and Bound showed to be sometimes better in small and niche problems but can also be better in other variations of 0-1 Knapsack where Dynamic Programming may fall behind or not work at all.