Jiaqing Mo
CPE 349
Final Project Report: 0-1 Knapsack Problem

	easy20	easy50	hard50	easy200	hard200
Enum	0.615 s	-	-	-	-
Greedy	0.151 s	0.149 s	0.144 s	0.158 s	0.171 s
Dyn Prog	0.140 s	0.156 s	0.171 s	0.194 s	0.420 s
B' n B'	0.148 s	3.663 s	3.213 s	(> 1 hour) out of memory	(> 5 hours) out of memory

One advantage for brute force is that it is easy to implement, so if the client doesn't have a lot of time to implement or doesn't have much money to hire professional to implement, the brute force is a good choice. Also, it works well for small size of input. If the client is to use this algorithm for one or two times and the problem size is small (less than 20), it is a good choice.

One of the cons of brute force implementation is that if the problem size get large, it takes a long time to finish, since the time complexity is $O(2^n)$, where n is the number of items. Also, this can work for both integer and double weights.

One advantage of greedy algorithm is that it takes the shortest time to finish with time complexity of O(n), where n is the input size. And when the input size is large and the client expect short time to finish, while the exact result isn't important (the approximate result can work too), the best choice would be the greedy approach. Also, it can take double as item weights.

Greedy algorithm for 0-1 knapsack problem doesn't always give an optimal solution, as it always picks up the highest density item that can fit in the knapsack. My criterion to sort the items in order of the decreasing value divided by the weight.

Dynamic solution gives an exact solution to the 0-1 knapsack problem, and it takes a shorter time to finish with time complexity O(mn), where n is the number of items and m is the capacity. If the client has a lot of memory and wants a shorter running time to get an opitimized solution, then dynamic solution is a better choice.

The dynamic solution only works for positive and integer weights, as the table's index represent each capacity. Also, when the capacity become large, the table size will become large, which can take a lot of memory.

Branch and bound also can give an exact solution, it uses less memory than dynamic programming, because each node is free after their children are put in the queue. If the client wants to use less memory to get an optimized solution, but don't care a lot about the running time, branch and bound is a better choice. Another advantage is that branch and bound can take both integer and double weights. Also, the time complexity is not affected by the capacity. So if the problem has a large capacity, compared to the dynamic solution, branch and bound is a better choice.

However, branch and bound in the worst case can have a time complexity of O(2^n), it can happen when the last item has a heavy weight, and all solutions can't be determined until the leaves of the tree. But it may take a shorter time to finish if a heavy item is computed early. As a result, the running time for branch and bound is unpredictable, especially when the input size is large. So it is not a good choice if the client want a predictable running time. At the same time, branch and bound may take a long time for the programmers to develop.

To improve the branch and bound solution, I can save all the parent nodes of all the live nodes, and have a parent pointer for each node. So I don't have to copy every array from the parent to the child, because the arrays are not necessary. And I can trace back after a find solution at the leave. As a tradeoff, this can take a lot of memory when there are a lot live nodes. But the time to finish the program will be a lot faster. But this will still be slower than the dynamic solution. At the same time, I can calculate a more precise bound by adding the rest items using greedy algorithm, which can be a better pruning.

My brute force solution can be improved by processing the binary string after it is generated, instead of storing all of the binary string and process them after. This can use less memory, but will take a longer time to compute.