In this project, we were asked to solve the knapsack problem in three ways: once by using an exhaustive search method, once by using dynamic programming, and once by using a method of our choosing.

Firstly, the time and space efficiencies for each implementation will be proven:

Exhaustive search:

```
def exhaustive(items, capacity):
    possibleSolutions = 2 ** len(items)

    max_value = -1
    optimal_items = []

    for i in range(possibleSolutions):
        decision_matrix = [int(d) for d in str(bin(i)[2:]).zfill(len(items))]

        current_weight = 0
        current_value = 0

        for j in range(len(decision_matrix)):
            if decision_matrix[j] == 1:
                current_weight += items[j][0]
                current_value += items[j][1]

        if current_weight <= capacity and current_value > max_value:
            max_value = current_value
            optimal_items = decision_matrix

    return optimal_items
```

In the exhaustive search, all possible subsets were generated and compared against each other to see which would give an optimal solution.

Because this assignment deals with a 0/1 knapsack, then all of the possible subsets for an item set of *n* items would be the power set of *n*: $\wp(n)$.

As $|\wp(n)| = 2^n$, that means that the outermost loop is $O(2^n)$. However, because this particular solution was implemented by getting $\wp(n)$ via a binary representation, the process of using the binary vector to check each solution is O(n), as shown in the inner loop. Therefore, the time efficiency for the exhaustive search is $O(n) * O(2^n) = O(n2^n)$.

As for space efficiency, the only storage that changes with *n* is `optimal_items` and `decision_matrix`. Therefore, the space efficiency is $O(n) + O(n) = O(2n) = O(n)$.

Dynamic Programming:

```
def dynamic(items, capacity):
    memoization_table =
     [[0 for i in range(capacity + 1)] for j in range(len(items) + 1)]

    for i in range(1, len(items) + 1):
        current_item = i - 1

        for j in range(capacity + 1):
            if j - items[current_item][0] >= 0:
                if items[current_item][1] +
                 memoization_table[i - 1][j - items[current_item][0]] >
                 memoization_table[i - 1][j]:
                    memoization_table[i][j] =
                     items[current_item][1] +
                     memoization_table[i - 1][j - items[current_item][0]]
                else:
                    memoization_table[i][j] = memoization_table[i - 1][j]

            else:
                memoization_table[i][j] = memoization_table[i - 1][j]

    i = len(items)
    j = capacity
    optimal_items = [0 for x in range(len(items))]

    while i > 0 and j > 0:
        if memoization_table[i][j] != memoization_table[i - 1][j]:
            optimal_items[i - 1] = 1
            j -= items[i - 1][0]

        i -= 1
```

In the dynamic programming approach, a table which represents the subproblems is created. Because the table has dimensions of *n* and capacity *c*, then the time efficiency to create and fill in the table is $O(n) * O(c) = O(nc)$. This creation and generation of the table is presented by the nested for loops.

The other major aspect of this implementation is the backtracking algorithm. The specific way that this algorithm was implemented was that the program started in the bottom right hand corner of the `memoization_table`, go up until the next number is different (thus giving you the smallest weight for the largest value), then move over to the column where the $capcaity = capacity_{init} - weight$. The process is then repeated, until all the solutions are found.

Even though the size of the matrix is O(nc), there is no possible way that the program will iterate through every cell, simple due to the nature of the algorithm. Therefore, the time efficiency for the backtracking is always $<$ O(nc), and therefore can be ignored.

Hence, the time efficiency for the dynamic programming approach is $O(nc)$.

As mentioned previously, the `memoization_table`, is a table of dimensions *n* x *c*. The only other data storage in this approach is `optimal_items`, which is of size *n*. Therefore, the space efficiency for the dynamic programming approach is $O(nc) + O(n) = O(nc)$

Greedy Algorithm:

```
def greedy(items, capacity):
    complete_array = []

    for i in range(len(items)):
        complete_array.append([i,
                               items[i][0],
                               items[i][1],
                               items[i][1] / items[i][0]])

    complete_array = sorted(complete_array, key=lambda x: x[3])

    remaining_space = capacity
    optimal_items = [0 for x in range(len(items))]
    i = len(complete_array) - 1

    while remaining_space > 0 and i >= 0:
        if remaining_space - complete_array[i][1] >= 0:
            optimal_items[complete_array[i][0]] = 1
            remaining_space -= complete_array[i][1]

        i-= 1
```

The approach that I chose for this program is a greedy algorithm. The basic idea behind the algorithm is to sort the items based on the value to weight ratio. Then, the algorithm will try to put the item with the largest value to weight ratio until no more items can fit.

There are three main aspects of this algorithm: generating the value to weight ratios, sorting the algorithm, and then determining what are the optimal values for the knapsack.

For the generation of the value to weight ratios, the time complexity is fairly straightforward: O(n).
For the sorting of the algorithm, python's default Timsort was used, which has a time complexity of O(nlogn)
For the determination of what are the optimal values for the knapsack, it is obvious that the worst case is if all the items belong in the knapsack, which would yield a time complexity of O(n).

Therefore, the total time complexity for the greedy algorithm is $O(n) + O(n) + O(nlog\ n) = O(n \log n)$.

As for the space efficiency, there are only two instances of data storage, `complete_array` and `optimal_items`. The space efficiency for both is O(n), so the overall space efficiency for the greedy algorithm is O(n).

Do all three algorithms provide an optimal solution?
Yes.

For the exhaustive solution:
This is fairly obvious. Because it generates and compares all possible solutions, it is guaranteed to find an optimal solution.

For the dynamic programming solution:
The way that the DP algorithm works is by simplifying the problem to a set of smaller subproblems. In this case, the sub problem is a simple matter of choosing if the previous sum of items or the current item is part of the optimal solution. By that recursive logic, by the end of program, the solution to the overall problem will be an optimal solution.

For the greedy solution:
Because the items are sorted due to the value to weight ratio, they are literally sorted in terms of which item is more optimal. Therefore, when the algorithm is iterating through the knapsack, it will always pick the most optimal choice, and yield an optimal solution.

Which is the best solution? Is this true for all knapsack examples?
For this situation, the greedy solution is the best solution. This is because its time and space complexity is less than both of the other solutions.

This is also true for all knapsack examples, including non-0/1 knapsacks. This is because the greedy algorithm puts the "best bang for the buck" items in the knapsack, which means that it will always end up giving an optimal solution.

Other data sets
To ensure that these algorithms work on more than just the supplied dataset, two more datasets were chosen to run the algorithms.

Dataset 1
Capacity: 10
Weight data set: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Value data set: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Results:
Exhaustive search solution: 10 optimal subset: [[10, 10]]
Dynamic search solution: 10 optimal subset: [[1, 1], [2, 2], [3, 3], [4, 4]]
Chosen(greedy) search solution: 10 optimal subset: [[10, 10]]

This dataset was chosen because there are multiple optimal solutions to this specific data set. This demonstrates that the different solutions are capable of getting *an* optimal solution, regardless of how many there may or may not be.

Dataset 2
Capacity: 100
Weight data set: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29
Value data set: 4, 5, 8, 8, 7, 12, 6, 11, 10, 8, 12, 13, 15, 12, 10, 18, 11, 18, 25, 17, 24, 19, 28, 23, 18, 20, 31, 32, 20

Results:
Dynamic search solution: 141 optimal subset: [[1, 4], [2, 5], [3, 8], [4, 8], [5, 7], [6, 12], [8, 11], [13, 15], [16, 18], [19, 25], [23, 28]]
Chosen(greedy) search solution: 141 optimal subset: [[1, 4], [2, 5], [3, 8], [4, 8], [5, 7], [6, 12], [8, 11], [13, 15], [16, 18], [19, 25], [23, 28]]

This dataset was chosen to ensure that the exhaustive method was essentially unusable for a decently large dataset. With the above data, the exhaustive method hadn't completed after 3 minutes of execution time, because there are 2^{29} combinations to search: confirming the theory behind the time complexities with practice.