CS 333 PROJECT 4 REPORT

1. Implementation Details:

The implementation of this code is fairly simple, as it only consists of a single Java class called `Main` with a `main` method and a `solve` method. The `main` method reads input from the user, which consists of the number of items `n`, the weights and profits of the items, the maximum weight that can be carried `b`, and the minimum profit required `m`. It then calls the `solve` method to find a solution to the Knapsack problem.

The `solve` method is a recursive function that performs an exhaustive search to find a subset of items with total weight at most `b` and corresponding profit at least `m`. It takes as input the current item being considered `i`, the current total weight `weight`, the current total profit `profit`, the weights and profits of all the items, the maximum weight that can be carried `b`, and the minimum profit required `m`. It returns a boolean value indicating whether a solution has been found or not.

The `solve` method works by considering each item one at a time, and either including it or not in the subset. It does this by making two recursive calls, one in which the current item is included and one in which it is not. If all items have been considered and the total weight is at most `b` and the corresponding profit is at least `m`, then the function returns `true`, otherwise it returns `false`.

The `main` method also includes a loop that allows the user to perform multiple searches, and a try-catch block to handle any exceptions that may occur while reading the input.

2. Exhaustive Search Method:

The `solve` method is a recursive function that performs an exhaustive search to find a subset of items with total weight at most `b` and corresponding profit at least `m`. It does this by considering each item one at a time, and either including it or not in the subset.

The method has the following pseudocode:

```
solve(i, weight, profit, w, p, b, m)
    if i == length of w:
        if weight <= b and profit >= m:
            return true
        else:
            return false
    if solve(i+1, weight, profit, w, p, b, m) == true:
        return true
    return solve(i+1, weight + w[i], profit + p[i], w, p, b, m)
```

The base case of the recursion is when `i` is equal to the length of the array `w`, which represents the number of items. In this case, the method checks if the total weight is at most `b` and the corresponding

profit is at least `m`. If both conditions are true, it returns `true`, otherwise it returns `false`.

If `i` is not equal to the length of `w`, the method makes two recursive calls. The first call is made with the same values of `i`, `weight`, and `profit`, which corresponds to not including the current item in the subset. The second call is made with `i` incremented by 1, `weight` increased by the weight of the current item, and `profit` increased by the profit of the current item, which corresponds to including the current item in the subset. If either of these recursive calls returns `true`, the method returns `true`, otherwise it returns `false`.

3. Time Complexity Analysis:

The time complexity of the `solve` method can be analyzed with respect to the number of items `n`. In each recursive call, the value of `i` is incremented by 1, and the method makes two recursive calls. Therefore, the total number of recursive calls is equal to twice the number of items.

The time complexity of the method is therefore `O(2^n)`, since the number of recursive calls grows exponentially with the number of items. This means that the time required to find a solution increases rapidly as the number of items increases.

For example, if there are 10 items, the method will make 1024 recursive calls (2^10). If there are 20 items, the method will make 1048576 recursive calls (2^20). As the number of items increases, the time required to find a solution will become impractical for all but the smallest instances of the problem.

In general, exhaustive search methods are only practical for small instances of a problem, since the time required to find a solution grows exponentially with the size of the input. In cases where the problem size is large, more efficient algorithms, such as dynamic programming or greedy algorithms, should be used instead.