0/1 KNAPSACK PROBLEM

Group 1 CSD302 FALL2022 FPTU

September, 2022

I. INTRODUCTION

One of the classic combinatorial optimization problems is the knapsack problem which finds many applications in the computing domain. From a given set of items (each with a weight and a value), the objective is to find the number of items that can be included so that the total weight is less than or equal to a given limit and also the total value is the maximum. The Knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "Knapsack problem" dates back to the early works of the mathematician Tobias Dantzig (1884–1956) and refers to the commonplace problem of packing the most valuable or useful items without overloading the luggage.

1. Applications

The knapsack problems have a variety of real-life applications including financial modeling, production and inventory management systems, stratified sampling, design of queuing network models in manufacturing, and control of traffic overload in telecommunication systems. Other areas of applications include yield management for airlines, hotels, and rental agencies, college admissions, quality adaptation and admission control for interactive multimedia systems, cargo loading, capital budgeting, cutting stock problems, and computer processing allocations in huge distributed systems.

2. Research Objectives

We consider a generalization of the 0-1 knapsack problem. We mention some basic and advanced algorithmic ideas: Brute force, Meet in the middle, Memorization, Dynamic programming, Greedy, and Branch and Bound. For each formulation, we design an efficient program to compute and compare the strength-derived specific results for each algorithm.

II. PROBLEM DEFINITION

The knapsack problem is an optimization problem used to illustrate both problem and the solution. It derives its name from a scenario where one is constrained in the number of items that can be placed inside a fixed-size knapsack. Given a set of n items with specific weights W[] and values P[], the aim is to get as much value into the knapsack as possible given the capacity C of the knapsack. In addition, there are Qi copies of item i available, where quantity Qi is a positive integer satisfying $1 \leq Qi \leq \infty$. If Qi = 1 for i = 1, 2, …, n, the problem is a 0-1 knapsack problem.

The binary decision variables xi, with xi = 1 if item i is selected, and xi = 0 otherwise. The goal is to:

Maximize $\sum_{i=1}^{N} Pi.xi$

Subject to the constraints $\sum_{i=1}^{N} Wi.xi \leq C$ And $xi \in \{0, 1\}$

Input and output description

Standardize the problem with the function Knapsack(C, n, W, P) form for the whole algorithm.

		Data type	Description
Input	C	Integer	Capacity of knapsack
	n	Integer	Number of items
	W	Array of integer	Weight associated with each item
	P	Array of integer	Price associated with each item
Output	result	Array of integer	list of items that have the total maximum price can be contained in the knapsack

***Note:** the output of the algorithms in this study will be an integer describing the maximum price that can be contained in the knapsack, the purpose is to simplify the algorithms for simple description and help easier for readers to understand.

III. DATA PREPROCESSING

1. Data Observation

i) Easy to observe: the items having a weight greater than capacity need to be removed first to reduce the number of calculations. These items will not be the optimal solution because they do not satisfy the constraints of the knapsack problem.

ii) Items with weight Wi appear only up to x = floor(C / Wi) times. So in the optimal solution, there can only be at most x items of mass wi. In the preprocessing step we will choose x items with the weight wi and price pi such that the sum of pi is maximum.

iii) More broadly, items with the same weight often have roughly the same value, so instead of considering items of weight wi we consider all items of weight in Wj such that C / Wj = C / Wi = x (x > 0). Similarly, we also choose only x items with the largest value.

2. Discussion

Handling (i), (ii) always ensures that the solution found is optimal and reduces the size of the problem in many cases.

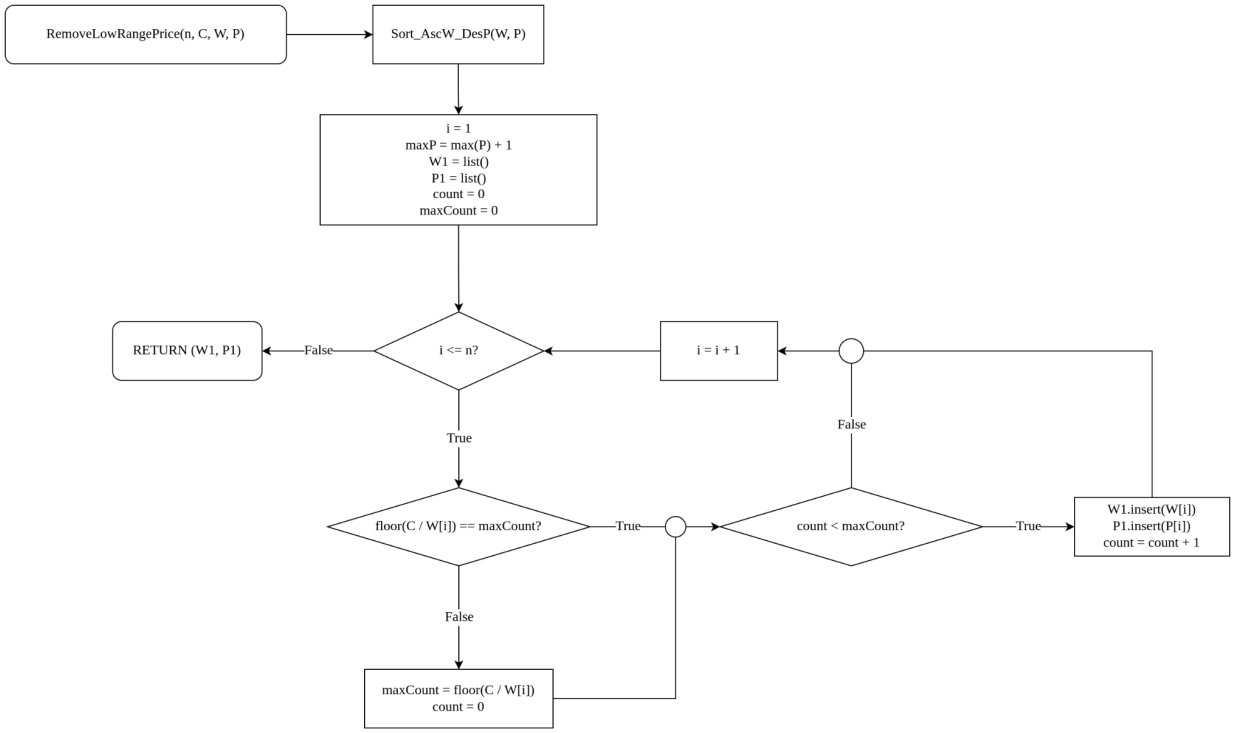
Process (iii) will also reduce more size of the problem more than (ii), but the problem result may not be optimal. Approximately, the solution will be close to the optimal solution, so our report will use this method for preprocessing data.

3. Data preprocessing

Based on the observation of the data we preprocess the input data. Following below steps:

- Sort items in ascending order based on the value a = Wi - Pi / (max(P) + 1). We can guarantee that items with the same Wi will be sorted in ascending order, while items with the same weight will also be sorted in descending order based on their price.
- **Method 1:** For each weight wi, keep only x = floor(C / Wi) items with the highest price into the knapsack.
- **Method 2:** For each item has the weight wi and another has the weight Wj satisfying C / Wj = C / Wi = x, only the x item with the highest value is kept. Accepting errors in this method.

Below is flowchart for method 2:



Time Complexity: O(n.log(n))

The time complexity for sorting by any advanced algorithm (like Quick sort, Merge sort,...) is O(n.log(n))

Auxiliary Space: O(1)

Because of using input to store sorted items.

→The size of the problem can be extremely reduced in the case that many items have roughly the same weight.

IV. ALGORITHMS

Many techniques have been developed to find the optimal solution to the knapsack problem. In this paper we classify these methods into 4 strategies:

(1) Brute Force and variants thereof

(2) Dynamic programming

(3) Greedy

(4) Branch and bound

1. Brute Force

A simple solution is to consider all subsets of items and calculate the total weight and price of all subsets. Consider the only subsets whose total weight is smaller than C. From all such subsets, pick the maximum price subset.

Optimal Substructure: To consider all subsets of items, there can be two cases for every item.

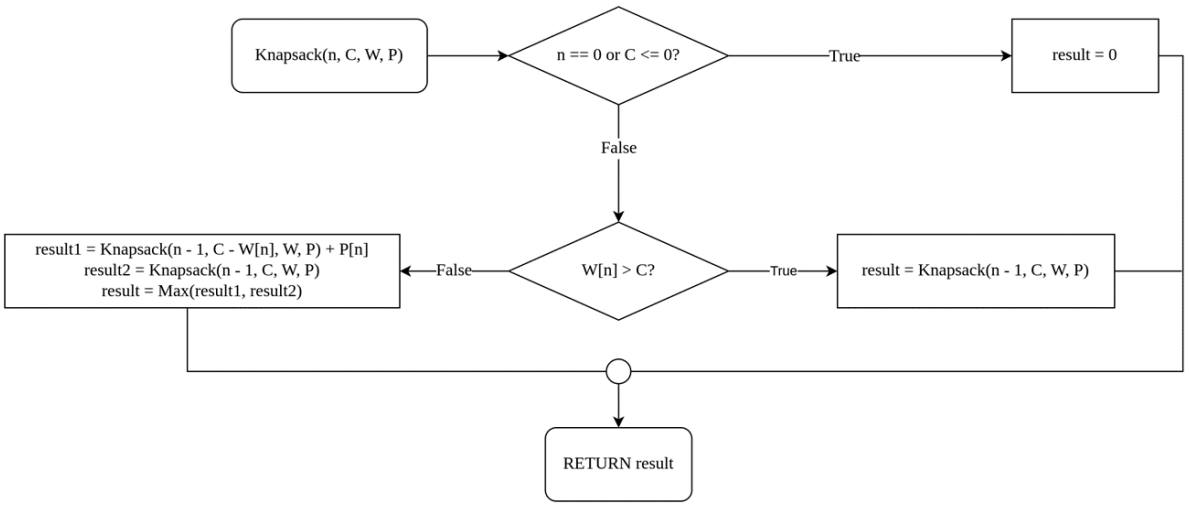
- **Case 1:** The item is included in the optimal subset.
- **Case 2:** The item is not included in the optimal subset.

Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values:

- Maximum value obtained by n-1 items and C weight (excluding nth item).
- Value of nth item plus maximum value obtained by n-1 items and C minus the weight of the nth item (including nth item).

If the weight of 'nth' item is greater than 'C', then the nth item cannot be included and Case 1 is the only possibility. Below is the flowchart for Brute Force implementation.

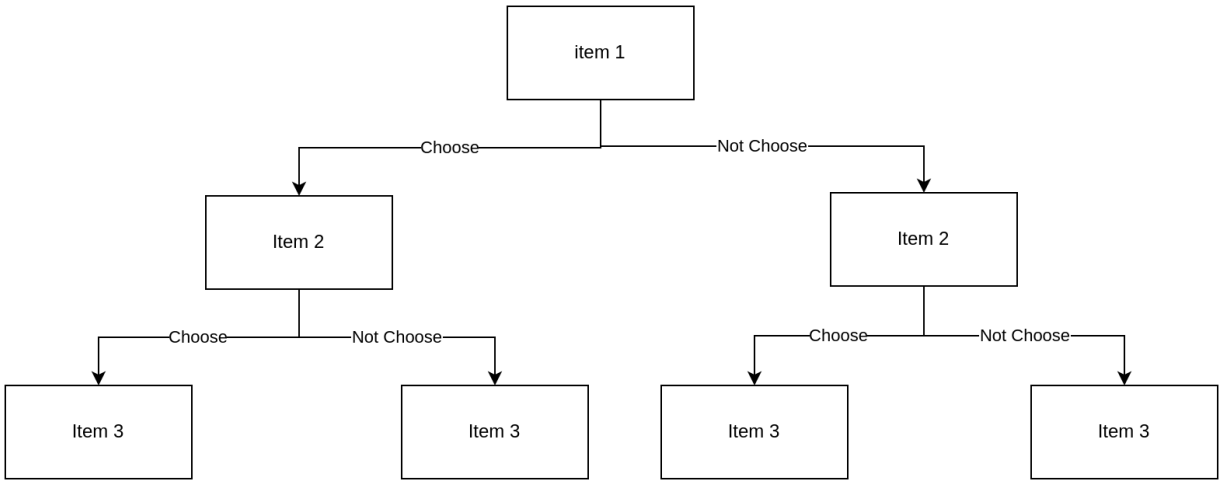
Flowchart:



Complexity Analysis:

Time Complexity: $O(2^n)$.

Each node of the recursion tree contains 2 children: choose and not choose the next item, we must consider all leaf nodes of the recursion tree with a height equal to n. Therefore, the time complexity of the Brute Force algorithm is $O(2^n)$.



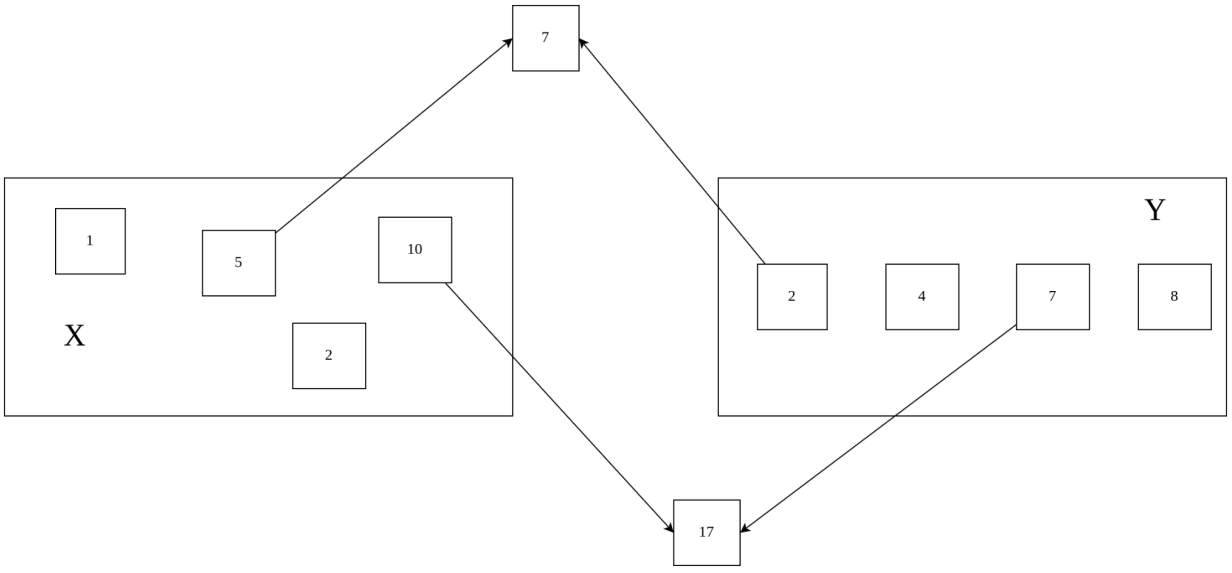
Auxiliary Space: O(n).

Because no extra data structure has been used for storing values but maximum O(n) auxiliary stack space(ASS) has been used for recursion stack.

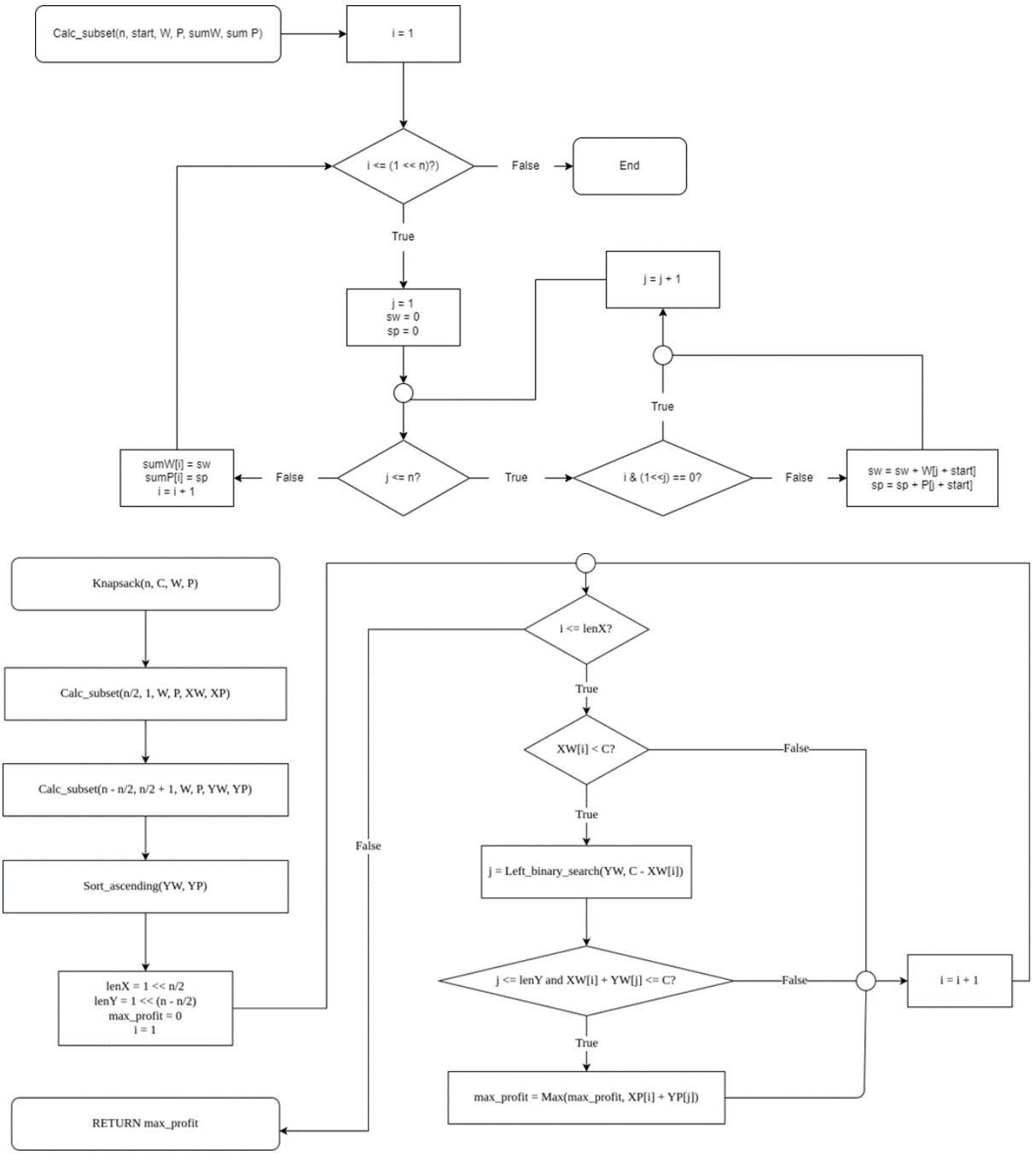
2. Meet in the middle

Meet in the middle is a search technique that is used when the number of items is small but not as small as the Brute Force algorithm can be used. Steps for solving the Knapsack problem using the Meet in the middle algorithm.

- Partition the set {1...n} into two sets A and B of approximately equal size.
- Compute the weights and prices of all subsets of each set
- For each subset of A, find the subset of B of greatest price such that the combined weight is less than C. Keep track of the greatest combined price seen so far.



Flowchart:



Complexity Analysis:

Time Complexity: O(n.$2^{n/2}$).

Simply iterate over all elements of array Y for each element of array X to check the existence of such a combination. This will take O($(2^{n/2})^2$) which is equivalent to O(2^n). To make it less complex, first sort array Y and then iterate over each element of X and for each element x in X use binary search to find maximum element y in Y such that x + y <= S.

Binary search here helps in reducing complexity from 2^n to $2^{n/2}.\log(2^{n/2})$ which is equivalent to $n.2^{n/2}$.

Auxiliary Space: $O(2^{n/2})$.

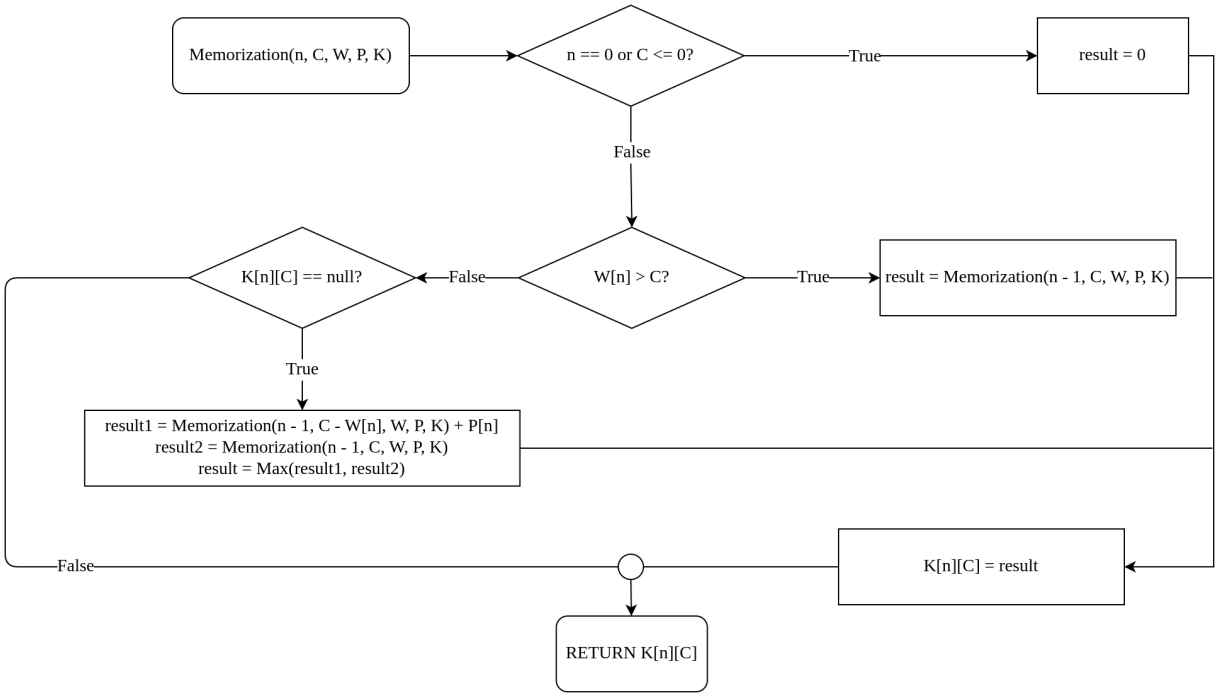
Because we use $4.2^{n/2}$ for contains all possible subsets of halves of the weight set (XW, YW) and price set (XP, YP).

In real-life implementation, we need an extremely large space to contain all possible items of all possible subsets. Therefore, in this report, we use this algorithm for comparing time complexity to other algorithms. The Meet in the middle algorithm is not used to find optimal items in real life.

3. Brute Force using Memorization technique

This method uses the Brute Force algorithm with the Memorization Technique. It is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. Now if we come across the same state (n, C) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time. This method gives an edge over the normal Brute Force approach in this aspect.

Flowchart:



Complexity Analysis:

Time Complexity: O(n*C).

Each node (n, C) in the recursion tree is calculated maximum once. Hence, the maximum calculations for all nodes are n*C. The redundant calculations of states are avoided, so the time complexity is O(n*C).

Auxiliary Space: O(n*C) + O(n).

The use of 2D array data structure for storing intermediate states and O(n) auxiliary stack space(ASS) has been used for recursion stack.

4. Dynamic Programming

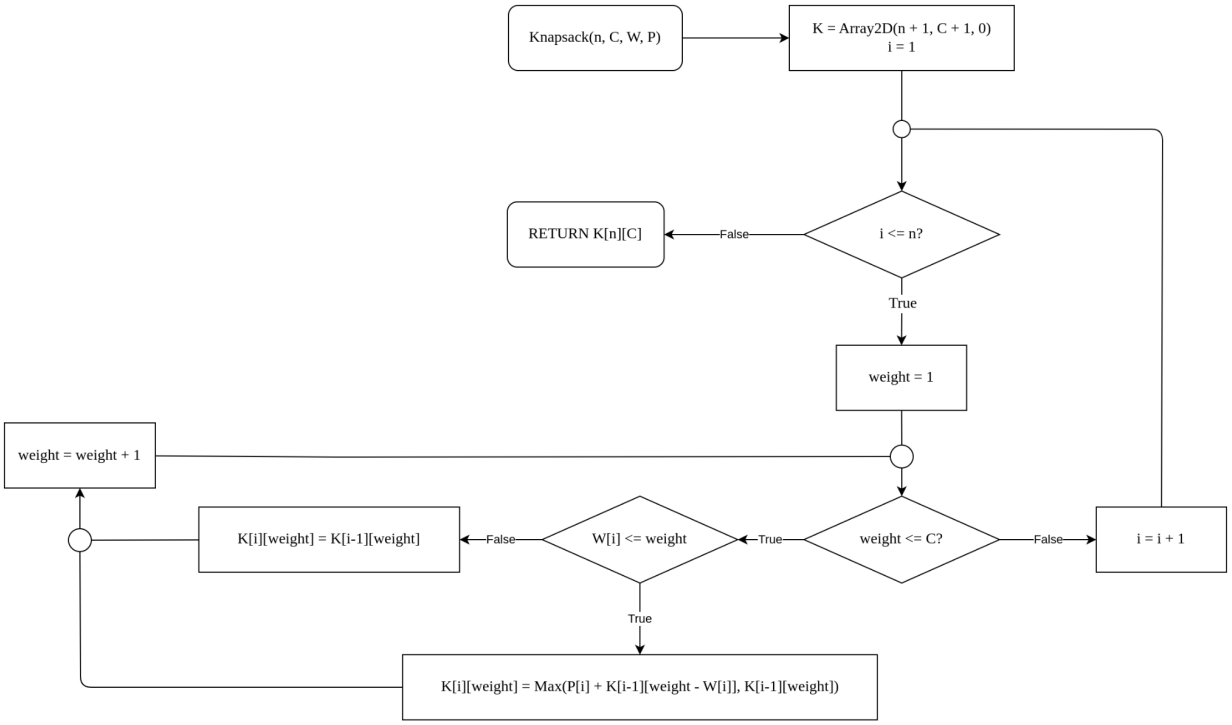
Dynamic Programming is a technique for solving problems whose solutions satisfy recurrence relations with overlapping subproblems. Dynamic Programming solves each of the smaller subproblems only once and records the results in a table rather than solving overlapping subproblems over and over again. The table is then used to obtain a solution to the original problem. The classical dynamic programming approach works bottom-up [1].

To design a dynamic programming algorithm for the 0/1 Knapsack problem, we first need to create a DP[][] table then consider all the possible weights from ‘1’ to ‘C’ as the columns and weights that can be kept as the rows.

The state DP[i][j] will denote the maximum price of ‘j-weight’ considering all items from ‘1 to ith’. So if we consider ‘Wi’ (weight in ‘ith’ row) we can fill it in all columns which have ‘weight > Wi’. Now two possibilities can take place:

- Fill ‘Wi’ in the given column.
- Do not fill ‘Wi’ in the given column.

Flowchart:



Complexity Analysis:

Time Complexity: O(n*C).

As for every weight element we traverse through all weight capacities 1<=w<=C.

Auxiliary Space: O(n*C).

Because we used 2-D array of size n*C to contain values in the table.

5. Greedy

Greedy programming techniques are used in optimization problems. A greedy algorithm makes a locally optimal choice in the hope that this will lead to a globally optimal solution. Clearly, greedy algorithms do not always yield the optimal solution [2].

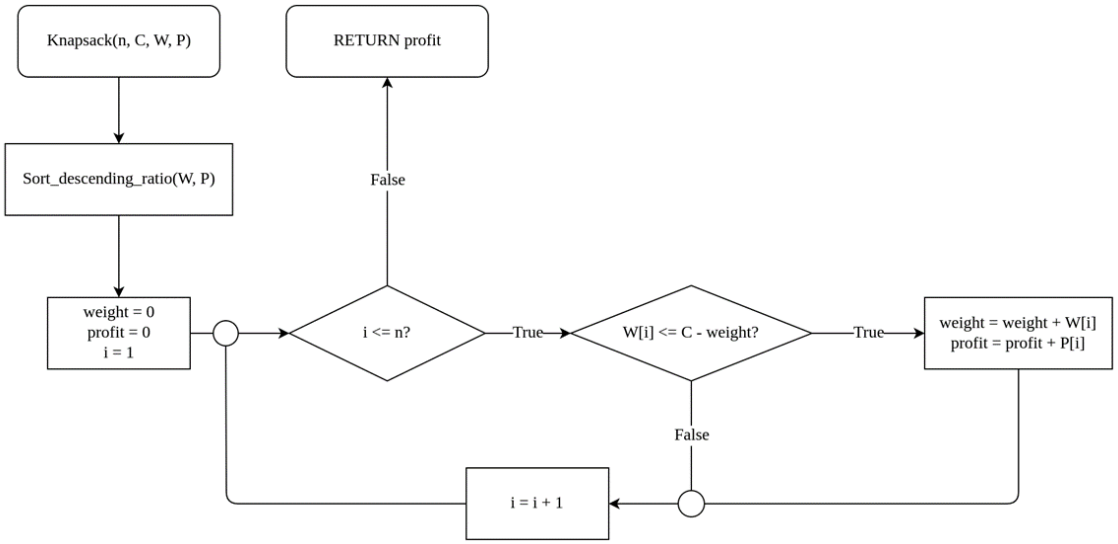
Possible greedy strategies to the 0/1 Knapsack problem:

1. Choose the item that has the maximum value from the remaining items; this increases the price of the knapsack as quickly as possible.
2. Choose the lightest item from the remaining items which uses up capacity as slowly as possible allowing more items to be stuffed in the knapsack.
3. Choose the items with as high a price per weight as possible.

We implemented and tested all three of the strategies. We got the best results with the third strategy - choosing the items with as high price-to-weight ratios as possible.

Below is the flowchart for Greedy algorithm implementation.

Flowchart:



Complexity Analysis:

Time Complexity: O(n.log(n))

The time complexity for sorting by any advanced algorithm (like Quick sort, Merge sort,...) is O(n.log(n))

The complexity for selecting items by traversing all of them is O(n).

The total time complexity of the greedy algorithm is O(n.log(n)) + O(n) ≈ O(n.log(n)).

Auxiliary Space: O(1).

Storing sorted weights and prices in the input arrays: the complexity is O(1).

Storing sorted weights and prices in new arrays: the complexity is O(n).

In above flowchart, we store sorted items in the input arrays. Hence, the complexity in our algorithm is O(1).

6. Branch And Bound

Branch and bound is a technique used only to solve optimization problems. It is based on the construction of a ‘state space tree’. a state space tree is a rooted tree where each level represents a choice in the solution set (In the context of the 0-1 Knapsack problem, it is chosen or not chosen) and depends on the upper bound. Any possible solution is represented by a path from the root node to the leaf node. The root node represents no optimal solution.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons [1]:

- The price of the node's bound is not better than the price of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the price of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

In the state space tree, a branch going to the left indicates the inclusion of the next item while a branch to the right indicates its exclusion. In each node of the state space tree, we record the following information:

level	indicates which level is the node at
cumValue	the cumulative price of all items that have been selected on this branch
cumWeight	the cumulative weight of all items that have been selected on this branch
nodeBound	Upper bound of maximum profit in subtree of this node

We may determine the upper bound by adding the cumulative value of the items already selected in the subset, and the product of the knapsack's remaining capacity and the best per-unit payoff among the remaining items, which is Pi+1 / Wi+1.

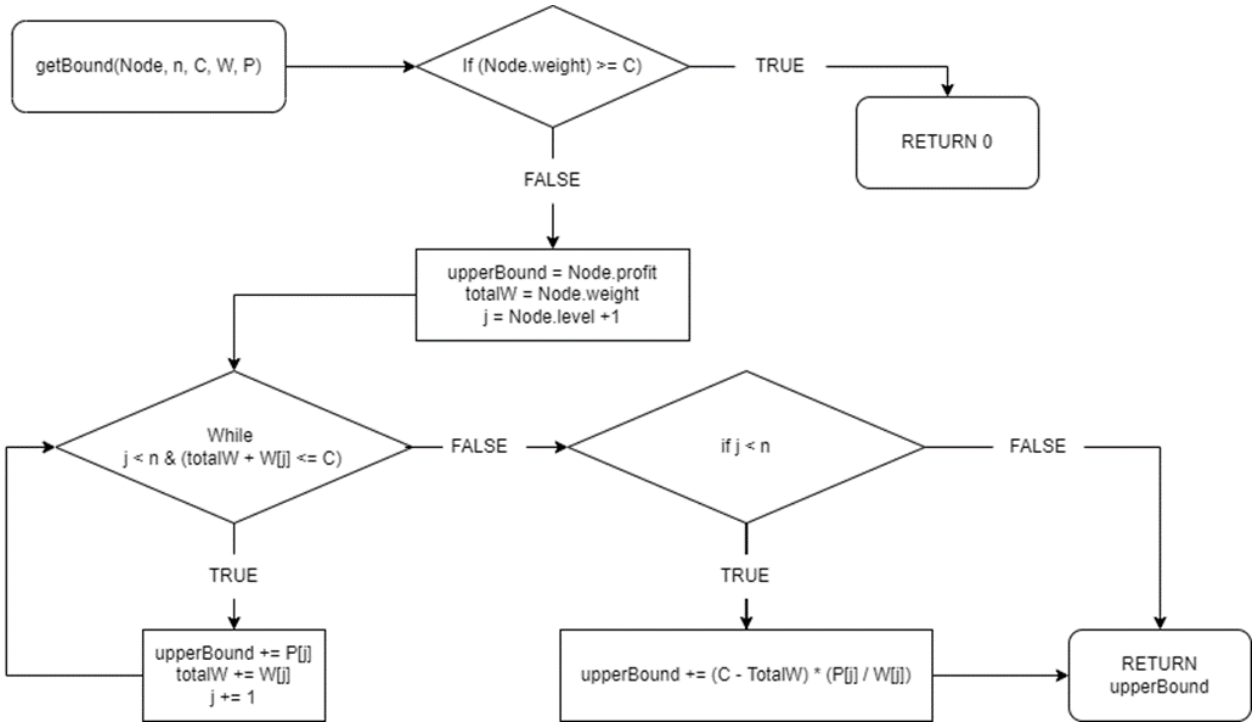
$$Upper\ Bound = P + (C - W).(P_{i+1}/W_{i+1})$$

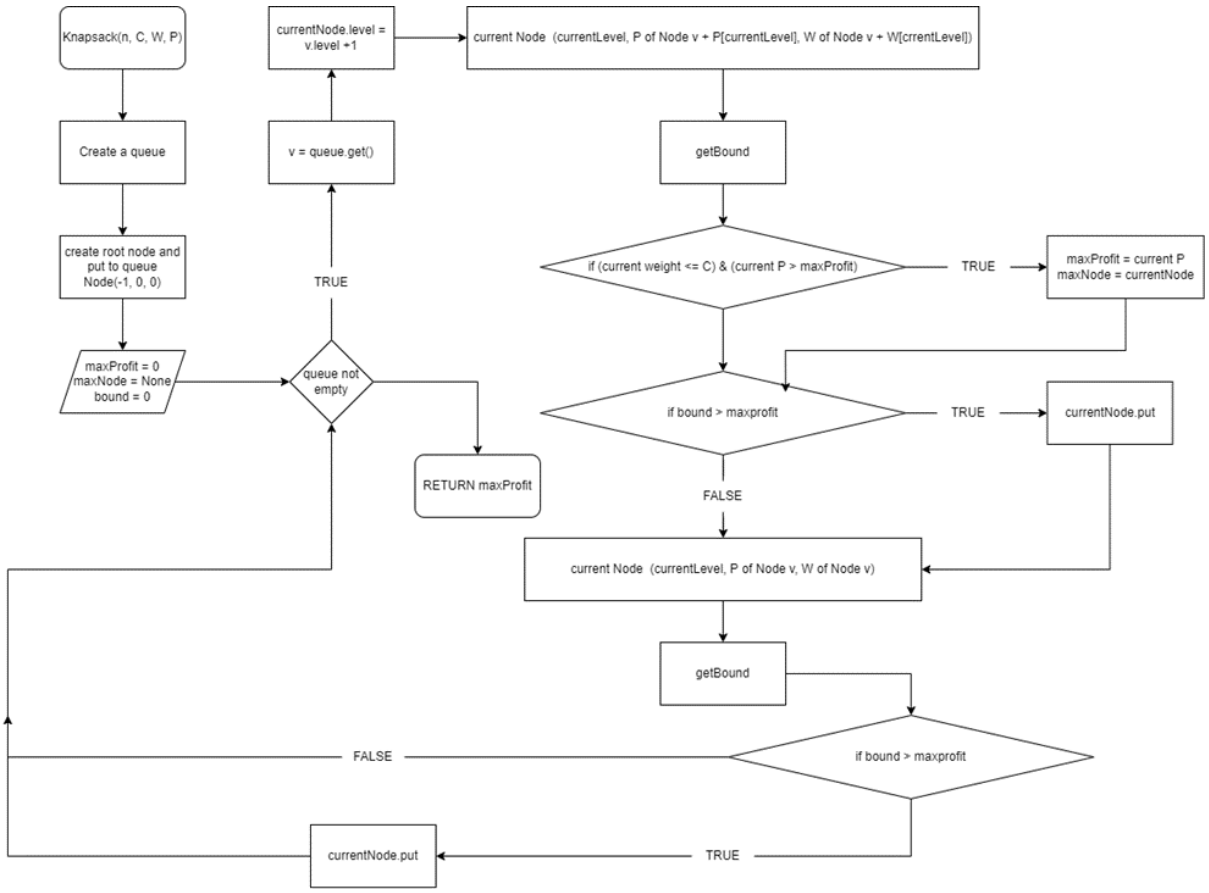
Initializing a state space tree costs a lot. To save more money on the problem of finding the maximum price, we use a queue to store the nodes. Step for solving Knapsack problem using Branch and Bound algorithm:

1. Sort all items in decreasing order of ratio of price per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.

4. Create a dummy node of the decision tree and enqueue it to Q. Profit and weight of the dummy node are 0.
5. Do following while Q is not empty.
    - Extract an item from Q. Let the extracted item be u.
    - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
    - Compute bound of next level node. If bound is more than maxProfit, then add the next level node to Q.
    - Consider the case when the next level node is not considered as part of the solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

Flowchart:





Complexity Analysis

Time Complexity:

- Worst case: $O(2^n)$
- Best case: O(n)

In the worst case, time complexity is still given by $O(2^n)$ in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to be explored, and hence its best case time complexity is given by O(n). Since this method requires the creation of the state space tree, its complexity will be exponential.

Auxiliary Space: The required memory depends on the length of the priority queue.

V. RESULTS AND DISCUSSION

For the testing of the different algorithms, we create two different datasets to measure running time. We performed two types of testing. For the first one, we increase the number of items to be considered and keep the capacity constant (C = 1000). The second tesing, we increase the capacity and keep the number of items to 20.

Testing I: Increase the number of items & Capacity = 1000

	Number of items (with C = 1000)	Brute Force	Meet in the middle	Brute Force with Memorization	Greedy	Branch and Bound	Dynamic Programing
0	5	0.0426	0.0388	351.9068	0.01812	0.1180	1.9712
1	10	0.7052	0.1871	1.3172	0.01049	0.3376	4.1101
2	20	13.1163	8.8043	8.8298	0.020266	0.9617	7.6656
3	40	NaN	17533.03	55.1559	0.027657	11.4786	15.311
4	100	NaN	NaN	225.1823	0.052214	3332.492	40.0424
5	500	NaN	NaN	2580.368	0.1761	NaN	217.778
6	1000	NaN	NaN	4667.795	0.3449	NaN	454.0722

Table 1

Algorithms with exponential complexity (Brute Force, Meet in the middle, Brand and Bound) will limit the number of items. The number of operations will be extremely large to handle as the number of items increases to perform the calculation of all cases. Overall, Meet in the middle has improved runtime compared to brute force. Brand and bounds by adding constraints (upper bound) to limit having to traverse the entire state space tree allow processing more items, and also demonstrates a speed improvement.

The greedy algorithm has the fastest running time among the considered algorithms. This is because the complexity depends on the step of sorting the items according to certain criteria. However, the greedy results in many cases do not match the optimal results when performing Brute force. This is a trade-off between the accuracy of the results and the execution time.

Testing II: Increase the capacity & Number of Items = 20

	Capacity (with N = 20)	Brute Force	Meet in the middle	Brute Force with Memorization	Greedy	Branch and Bound	Dynamic Programing
0	10	9.7608	8.1911	0.3130	0.0107	0.6260	0.0977
1	100	34.3627	8.2082	2.2003	0.0123	1.2829	0.7781
2	1000	13.1559	8.5003	9.0615	0.0188	1.0833	7.6012
3	10000	14.0898	8.2123	21.0273	0.0579	0.8614	79.9462
4	100000	102.6425	8.2767	303.1452	0.0216	6.4849	873.2309
5	100000	46.4916	8.4288	213.1922	0.0250	1.6725	7959.625

Table 2

In small capacity cases (using the number of items as a benchmark), dynamic programming will perform better than the rest of the algorithms, however, the efficiency will decrease as the knapsack capacity increases, because the time complexity of the recursion approach can be optimized to O(n*C). Same case for Brute Force with memorization. The remaining algorithms have stable execution times because the complexity does not depend on the capacity.

Testing with preprocessed data

Data preprocessing helps to reduce the size of the problem significantly compared to the original size. We will test the Brute Force algorithm (the algorithm that complexity is strongly influenced by the number of items) to test the improvement in time and efficiency when performing data preprocessing.

Preprocess	**New size**	**Time**	**Result**
Do not preprocessing	100	1627.6062	796
Remove large weight items	68	843.1954	796
Remove low price items	61	389.4210	796
Remove range of low price items	23	146.0216	796

Table 3: C=200, n=100, W_{max} = 300

In Table 3, all 3 methods reduce the size of the problem, thereby reducing the execution time significantly. The final result is still guaranteed to be the optimal result when removing large-weight and low-price items. Depending on the data, different data density distributions will give different efficiency.

Preprocess	New size	Time	Result
Do not preprocessing	50	7.0323	478
Remove large weight items	32	3.0035	478
Remove low price items	30	2.5843	478
Remove range of low price items	12	1.0035	**464**

Table 4: C=100, n=50, W_{max} = 150

In certain cases, the value of the result is not the optimal solution. However, this is still an approximation worth considering. This trade-off provides a short processing time and a result close to the optimal solution.

VI. CONCLUSION

- Comparative research between strategies (Brute Force and variants thereof, Greed, Dynamic programming, Brand and Bound) shows that despite the complexity of these algorithms being known, the nature of the problem they are applied to makes some of them more suitable than others.
- The best approximation for the 0/1 knapsack problem is dynamic programming. Although the algorithm depends on the number of items and the capacity of the knapsack, it will have difficulty in cases where the two parameters are too large. Dynamic programming is still recommended because these algorithms are easily programmed and understood by the user.
- Although the Greedy algorithm does not give the optimal solution, it returns a good approximation solution for the Knapsack Problem with small time complexity. In the case we do not need absolute solutions, the Greedy algorithm should be considered.
- Data preprocessing gives a good improvement to solving algorithms 01 knapsack problems. With the data observation and the allowed error, the selection of suitable data processing methods will give a highly applicable solution.

VII. REFERENCES

[1] Levitin, Anany. The Design and Analysis of Algorithms. New Jersey: Pearson Education Inc., 2003.

[2] Vince, Andrew. "A framework for the greedy algorithm." Discrete Applied Mathematics 121.1-3 (2002): 247-260.