Abstract: The Knapsack Problem has been a subject of fascination since its appearance in 1895, when Mathews' first laid the groundwork for combinatorial optimization problems like the Knapsack problem. The KP’s primary goal focuses on optimizing resource allocation by selecting the most valuable combination of items to fit within a knapsack's weight capacity. Over time, the KP has not only inspired the exploration of related problems such as the Traveling Salesman Problem and the Graph Coloring Problem, it has also exploded into a vast field of KP variants to address the practical complexities of real world uses. Notable problem variations include the 0/1 Knapsack Problem, Bounded Knapsack Problem, Unbounded Knapsack Problem. Additionally, two notable solution variations include the Extended Greedy Algorithm and Dynamic Programming Algorithm, which are the primary focus of this report. As the KP remains a compelling and universal problem, researchers and mathematicians continue to delve into new problem variations and discover additional solution variants; the significance and impact of the Knapsack Problem continues to command attention and drive innovation.

Keywords: Knapsack Problem, Resource Allocation, Unbounded, Bounded, Greedy, Dynamic Programming, 0/1, combinatorial optimization

# Introduction

Imagine a daring thief who has found himself amidst a legendary treasure trove, equipped with just a worn knapsack on his back. As his eyes dart across the most enticing items in the room, he is sorely reminded that his bag cannot withstand the weight of everything he wants. He finds himself forced to make strategic decisions - Should he go for the priceless artifact, the rare antique watch, or the ornate silver vase? They each have different weights and values. Imposed by the weight limit of his bag, the thief finds himself facing the same challenge as some of the greatest intellects for generations.

This modified version of the classic Thief example from Bhargava demonstrates an intriguing optimization challenge, known as the Knapsack Problem (KP), that has tested mathematicians, computer scientists, and even fictional thieves throughout history. At its core, the KP focuses on optimal resource allocation by framing it into one basic question: How can we maximize the value of the items in the knapsack without exceeding a weight limit?

# Problem Statement

The classic KP challenge revolves around the efficient packing of items into a knapsack with a limited weight capacity. The problem includes a knapsack with a predetermined weight capacity and a variety of items, each characterized by its corresponding weight and value. It is assumed that the weight capacity is at least as large as the heaviest item, and as many items as possible can be stored in the bag until the weight capacity is me. As the challenge focuses on distribution of the items, real world constraints such as dimensions and size are ignored [1].

While there are many variations of the KP, one of the most discussed and researched is known as the 0/1 KP. Some variations allow for fractions of items to be added to the bag in order to obtain the most value, while 0/1 KP asserts that each item can be either be entirely included in the knapsack or entirely excluded. It is important to note that while we often allude to items going in a knapsack, mathematically, this problem simplifies down to simply a "combination of binary decisions" [6]. For each item in the knapsack, it is either in the bag (1) or not in the bag (0). The duration of this report will be in regards to the 0/1 KP, hereto referred to simply as KP, unless otherwise specified.

## *Mathematical Representation*

In the formal representation of the KP, a combination of a given set of items, characterized by their corresponding weight () and value (), are selected to place in the knapsack, such that the value is maximized while the weight does not exceed the knapsack's capacity (). This representation can be expressed as follows:

(1)

where

* represents the profit/value of item
* corresponds to the weight of item
* denotes the knapsack's maximum weight capacity
* is a binary decision variable that takes the value if item is included in the knapsack and otherwise.

The function is used to represent the total value/profit achieved by selecting items with their corresponding profits () .The constraint ensures that the total weight of the selected items does not exceed the knapsack's capacity () by restricting the combination of items that can be placed in the knapsack. Since this is the 1/0 variation of the knapsack problem, is used to represent the selection () or exclusion () of items, enabling the optimization process to find the optimal combination of items that maximizes the total profit while adhering to the weight constraint [6].

## *History*

The KP has a long history, dating back centuries to its first appearance in 1895 [6]. Although the official name had yet existed at the time, Mathews' work on Partition Theory introduced the underlying connections to combinatorial optimization problems like the Knapsack problem. For example, Partition Theory’s focus on dividing or representing a number as a sum of positive integers introduced the notion of dividing the knapsack's capacity into parts (weights) to maximize the overall value. This was just the beginning and set the stage for future developments in the KP [6][4]. Combinatorial optimization is a branch of mathematics which deals with finding the best solution from a limited selection and a finite set of possible combinations. However, the KP is just one of many problems in this category; Other well-known combinatorial optimization problems include the Traveling Salesman Problem, the vehicle routing problem, and the Graph Coloring Problem, to name a few [11].

The introduction of the 0/1 variant of the KP in 1980 by Gallo, Hammer, and Simeone further expanded the problem's scope. The addition of the decision variables () brought an additional layer of complexity to the KP which opened new avenues for research and algorithmic exploration [9]. Since then, researchers have been working to improve the algorithm efficiency to various instances of the KP and its variants. Through which significant algorithmic advancements have been made, enabling the solution of “nearly all standard instances [of the KP] found in the literature” [7].

## *Importance*

The importance of the KP serves as a powerful tool across the fields of mathematics and computer science, whether to explore how algorithms work or how efficiently they operate [1], or in day-to-day practical applications, showcasing its versatility and utility in solving diverse challenges. Computer scientists can use the KP to help in program partitioning and task allocation, where it is important to prioritize critical tasks while optimizing computational resources. For more a more practical example, the KP can be used to help efficiently pack food for survival situations, with a goal to maximize nutrition while minimizing overall weight [5]. Additionally, the KP can help assist investors in maximizing their profits while adhering to their budget constraints. The KP proves invaluable to the lumber industry, where the KP contributes to the optimization of log cutting processes, maximizing the value obtained from each log while minimizing waste. In fact, while KP is often considered a ‘packing’ problem, it also sometimes referred to as a ‘cutting problem [6].

The versatility of the Knapsack is evident in many real-world applications, from efficiently packing food for survival to optimizing investment portfolios, loading cargo planes, and even cutting logs, the real world often presents additional complexities that the basic KP does not account for. From nutritional sustainability, investment risk tolerance, time constraints, market price variations, to other practical factors, there appeared a substantial “need for [an] extension of the basic knapsack model,” which led to the development of “various extensions and variations,” with each variant offering a solution to tackle a specific scenario [6]. These extensions further enhance the KP’s applicability and make it invaluable across various industries and applications.

## *Variants*

There have been many variations created to tackle the 0/1 KP. Two of the most notable variants that have existed since the start of KP are the Unbounded Knapsack Problem (UNP) and the Bounded Knapsack Problem (BNP). These two variants focus on the limitation of item repetition. In the UNP, item amounts are unlimited and can be repeated ad infinitum in the knapsack. This allows greater flexibility in item selection and the potential to achieve a higher total value, but increases the number of possible solutions. Alternatively, the BNP limits the amount of times each item can be placed inside the bag. Since each item appears an equal number of times, the primary focus is on the specific combination of the items. Thus, the solutions tend to be much simpler and more realistic, but the number of possible solutions is limited [6].

Amidst the BNP, the Greedy and Dynamic Programming variants remain prominent for solving the 0/1 KP. The Greedy approach focuses on locally optimal choices but disregards the overall consequences, resulting in efficient but approximate solutions. Alternatively, the Dynamic Programming variant breaks the problem into smaller subproblems and methodically calculates optimal solutions, ensuring a more accurate solution but less-efficient running time.

1. *Greedy Algorithm:* An algorithm is considered greedy if, it prioritizes immediate choices over evaluating multiple options to find the most optimal result [8]. As their main focus is on local gains rather than long-term consequences, they are often considered naïve and heuristic in nature. However, in the case of KP, the Greedy algorithm stands out as an intuitive method because it takes a seemingly obvious approach: assign a value-to-weight ratio for each item and fills the bag with items with the largest ratio until it reaches as close to max weight as possible [6]. Though more efficient and easier to understand and implement, this algorithm variation may not always yield the most optimal solution when compared to dynamic programming [8]. As such, it should only be used for scenarios where "good enough" solutions are acceptable, such as attempting to return change using the least amount of coins as possible, or when attempting a variant of the shortest path problem [6][8]. For this reason, the Greedy Variant is “not as broadly applicable as other algorithm design approaches (such as dynamic programming)” [8].
2. *Dynamic Programming:* Alternatively, the Dynamic Programming (DP) approach emerges as one of the best techniques for solving the challenging 0/1 KP [7]. DP works by breaking a larger problem down into smaller subproblems, and then solving them iteratively. Instead of dealing with an entire problem at once, DP starts at the most basic case and slowly works back up to the original problem. For KP, DP works in two phases, the forward phase and the backtracking phase. The forward phase is accomplished by employing the Bellman Recursion, which computes optimal solution values for knapsack subproblems [10][6]. In other words, the Bellman Recursion creates a grid of items and knapsack weights, through which it begins solving at the smallest weight capacity and works through each capacity and item, until it reaches the capacity of the original knapsack [2]. The backtracking phase starts from the final cell in the table, and identifies which items were included in each knapsack to achieve the maximum value. By efficiently breaking down the problem into smaller subproblems and iteratively computing the optimal solutions, the Dynamic Programming approach provides a reliable solution for the 0/1 Knapsack Problem with a worst-case guarantee on the running time [7].

# Algorithms

To better understand the previously discussed algorithm variations and how they work, the Greedy Algorithm and the Dynamic Algorithm, we will analyze the pseudocode of each. By analyzing the step-by-step procedures of these two approaches, we can visualize their methods for solving the Knapsack Problem.

## *Extended Greedy Algorithm*

Before observing the pseudo code of the Greedy Algorithm for KP, we must first discuss a special case which prevents the Greedy Algorithm from returning optimal solution [6]. Consider following values for our Knapsack instance:

Table I

|  |  |  |
| --- | --- | --- |
| Number of items | |  |
| Knapsack capacity | |  |
| Item 1 | weight |  |
| profit |  |
| efficiency |  |
| Item 2 | weight |  |
| profit |  |
| efficiency |  |

Given the values in Table 1, the Greedy Algorithm selects Item 1 due to its higher efficiency () compared to Item 2 (). Then, it places Item 1 into the knapsack, leaving a remaining capacity of 2. As no additional items fit within this space, the function concludes that the knapsack is full. However, the optimal solution in this case would have been to pack Item 2, which has a value of 3, resulting in a higher profit.

To address this limitation, I have implemented Kellerer’s "Extended Greedy" algorithm. This enhanced version incorporates an additional step to account for the possibility of choosing a single item with the highest profit value. This is accomplished by comparing the solution value obtained from the standard Greedy Algorithm to the largest individual profit value, of which the larger value is added to the knapsack. This modification will improve the algorithm's performance and ensure a more reliable and accurate approximation for the Knapsack problem.

Given the addition of this special case, the pseudocode for the Extended Greedy Algorithm:

|  |
| --- |
| * Initialize total weight and profit variables to 0 * Sort items by efficiency in decreasing order (highest first). * Iterate through the sorted items, and for each item:   + If adding the item does not exceed the weight capacity:     - Include the item.     - Update the total weight.     - Update the total profit.   + Else     - skip adding the item to the knapsack. |

With this improved approach, the Extended Greedy Algorithm aims to provide a promising alternative to the standard Greedy approach.

## *Dynamic Programming*

As previously mentioned, the Dynamic Programming approach solves the problem by breaking it down into smaller subproblems and iteratively finding each optimal solution. To illustrate the steps of the Dynamic Programming algorithm for the 0/1 Knapsack variant, consider the following pseudocode:

|  |
| --- |
| * Initialization   + Initialize total weight and profit variables to 0   + Create DP array[n+1, capacity+1] and initialize all cells to 0   *Note: The DP array will contain the Bellman Recursion Table*   * Bellman Recursion:   + iterate from 1 to n, For each item:     - iterate from 0 to capacity, For each sub-capacity:       * If Adding Current Item Exceeds Capacity         + Set cell to the previous best solution (cell above)       * Else         + Set current cell to max(cell above current cell, best value for remaining capacity) * Reconstruction   + Initialize empty list to store knapsack items   + While able to reverse iterate through DP table:     - If current cell is different from above cell:       * Include item in bag       * Update knapsack weight     - Continue to next item |

The Dynamic Programming algorithm considers all possible combinations of items to achieve the maximum profit while adhering to the knapsack's weight capacity. The iterative nature of Dynamic Programming guarantees the most optimal solution, making it a powerful algorithms for this solving this combinatorial optimization problem [6].

# Time Complexity

Now that the foundation for the Extended Greedy Algorithm and the Dynamic Programming Algorithm has been laid, we can now analyze the time complexity for the 0/1 Knapsack Problem.

## *Extended Greedy Algorithm*

There are two possible time efficiencies for the Extended Greedy Algorithm, depending on if the sort takes place inside the function or not. As mentioned previously, the Extended Greedy Algorithm sorts the items based on their efficiency in decreasing order. This sort requires , where pertains to the number of items. However, if we assume the items are sorted upon entering the greedy algorithm, the iteration through the sorted items takes linear time, [6]. During the iteration, all subsequent lines to verify if an item can be included in the knapsack and updating the total weight and profit variables can be done in constant time, . As such, the overall time complexity of the Extended Greedy Algorithm is if the sort takes place prior to the algorithm call, and if the sorting happens within the call. It is important to note that, no matter where the sorting takes place however, the program time complexity will be dominated by the sorting step and have a time complexity of , unless further code dominates the program.

## *Dynamic Programming*

Similar to the Extended Greedy Algorithm, There are two possible time efficiencies for the Dynamic Programming Algorithm, depending on included functions. Although the example I showed above includes a Reconstruction section, this is optional. If the purpose of the function is solely to find the most optimal value, it is possible for the algorithm to return after Bellman Recursion. The Reconstruction section backtracks through the DP table and returns the items within the knapsack as well[6].

The Bellman Recursion section of the Dynamic Programming Algorithm involves iterating through each item and each sub-capacity of the knapsack and filling a table to calculate their corresponding optimal value solutions. Each iteration runs in linear time, with the outer loop running once for each item from 1 to n+1, or , and the inner loop running once for each sub-capacity from to , or . This results in an overall time complexity of . All subsequent operations within the algorithm, including verifying if an item can be included in the knapsack and updating the total weight and profit variables can be done in constant time, . Therefore, if the algorithm were to return after the Bellman Recursion, the time complexity will be dominated by the second for-loop and have a time complexity of . Alternatively, if the Dynamic Programming Algorithm includes a Reconstruction section, there would be additional iterations, backwards through the table, significantly increasing the time complexity to .

# Conclusion

In conclusion, the Knapsack Problem (KP) has been a subject of interest since its first appearance in 1895, rooted in Mathews' work on combinatorial optimization. Over the years, it has paved the way for related problems like the Traveling Salesman Problem and the Graph Coloring Problem, becoming a worthy challenge in resource allocation. The KP's practicality has made it applicable in various real-world scenarios, leading to the exploration of variants and solutions such as the Bounded Knapsack Problem, Unbounded Knapsack Problem, 0/1 Knapsack Problem, Extended Greedy Algorithm, and Dynamic Programming Algorithm.The KP's versatility has made it applicable in diverse real-world scenarios, ranging from program partitioning, task allocation, and to investment portfolio optimization. As researchers and mathematicians continue to study new variations and algorithmic improvements, the KP's significance and popularity will continue to spread.

## Hurbans

* + Grokking Artificial Intelligence Algorithms
  + Rishal Hurbans
  + <https://learning.oreilly.com/library/view/grokking-artificial-intelligence/9781617296185/>

## Bhargava

* + Grokking Algorithms
  + Aditya Bhargava
  + <https://learning.oreilly.com/library/view/grokking-algorithms/9781617292231/>

## Geeks

## Mathews

* + Mathews, G. B. (25 June 1897). "On the partition of numbers" (PDF). Proceedings of the London Mathematical Society. 28: 486–490. doi:10.1112/plms/s1-28.1.486.
  + <https://books.google.com/books?id=DlJAAQAAIAAJ&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false>

## Rocca

* + Advanced Algorithms and Data Structures
  + By Marcello La Rocca
  + <https://learning.oreilly.com/library/view/advanced-algorithms-and/9781617295485/>

## Kellerer

* + Knapsack Problems
  + Hans Kellerer , Ulrich Pferschy , David Pisinger
  + <https://link.springer.com/book/10.1007/978-3-540-24777-7>

## Pisinger

* + Where are the hard knapsack problems?
  + David Pisinger
  + <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.7431&rep=rep1&type=pdf>

## Zingaro

* + Algorithmic Thinking
  + Dan Zingaro
  + <https://learning.oreilly.com/library/view/algorithmic-thinking/9781098128197/>

## Billionnet

* + Linear programming for the 0-1 quadratic knapsack problem
  + Alain Billionnet
  + On drive
  + <https://www.sciencedirect.com/science/article/pii/0377221794002290>

## Andonov

* + Unbounded knapsack problem: Dynamic programming revisited
  + <https://www.sciencedirect.com/science/article/pii/S0377221799002659>

## Wang

* + https://www.sciencedirect.com/science/article/abs/pii/S0020025522013627