# Abstract:

## Instructions:

* + - * “In a paragraph, give a summary of what the problem is trying to solve, what kinds of approaches there are to solving the problem (naive divide-and-conquer, dynamic programming, backtracking, etc.), and what kind of Big O time complexities it may have for those given approaches.”

## the problem is trying to solve

* + - a

## kinds of approaches there are to solving the problem

* + - a

## kind of Big O time complexities it may have for given approaches

* + - a

# Problem Statement:

## Instructions:

* + - “Describe what the knapsack problem is in detail. Describe its history and why its such an important problem in computer science. If there are any variants to the primary knapsack problem, shortly list and describe what the variant is and why it’s included.”

## What is the knapsack problem

* + - High Level
      * Modified Thief example from [Bhargava]
        + Imagine a thief attempting to steal valuable items from a treasure trove. He breaks in with only his knapsack, which has a weight limit of 5kg. He sees three items:

A priceless artifact [100k, 4kg]

a rare antique watch [80k, 2kg]

Ornate silver vase [90k, 3kg]

* + - * + He grabs most expensive item, A priceless artifact, and realizes he cannot carry any more and makes a break for it. However, as he is making his getaway, he realizes his mistake – while the single item has more value, it took up more than half of the weight. He should have grabbed the other two items, which would have been worth more collectively, while still being within the weight limit.
      * *While in this example, size and dimensions are not considered, ….*
      * “In the Knapsack Problem, a knapsack has a specific maximum weight that it can hold. Several items are available to be stored in the knapsack, and each item has a different weight and value. The goal is to fit as many items into the knapsack as possible so that the total value is maximized and the total weight does not exceed the knapsack’s limit. The physical size and dimensions of the items are ignored in the simplest variation of the problem” [Hurbans]
      * Has a set total weight capacity, and can hold any of the items
      * “the possibilities explode as the number of potential items increases.” [Hurbans]
      * “It will also be computationally expensive to try to brute-force every combination of items when the variables grow; thus, we look for algorithms that are efficient at finding a desirable solution.” [Hurbans]
      * Goal: to select items that provide the most value while ensuring that their total weight does not exceed the knapsack's capacity [kp]
    - Low Level [Kellerer]
      * A set of 'n' items, each characterized by their weight and value
      * For an item, j, these are denoted as weight ‘w\_j’ and value/profit ‘ p\_j’
      * The knapsack has a maximum weight capacity of 'c'.
      * The goal is to find a subset of items, ‘N’, to include in the bag such that the total value of included items is maximized without exceeding the weight capacity of the knapsack
      * This is mathematically defined as

|  |  |
| --- | --- |
|  | Profit of item |
|  | weight of item |
|  | a binary variable that indicates whether item 'j' is included or not |

## History

* + - has a long history of study dating back centuries
      * First appearance of Knapsack Problem in 1895 Mathews [Kellerer]
      * *While the official name had not yet been invented…*
      * *While this paper mainly focuses on ‘partition theory, which deals with dividing or representing a number as a sum of positive integers, it introduces the underlying connections to combinatorial optimization problems like the Knapsack problem. For example, In the Knapsack problem, the primary goal is to divide the knapsacks capacity into parts (weights) to maximize the value.*
        + combinatorial optimization a field of mathematics that deals with finding the best solution from a finite set of possible combinations or arrangements of discrete objects. Also seen in other problems including: [NR]

The Traveling Salesman Problem

The Maximum Clique Problem:

The Graph Coloring Problem

* + - so decades of algorithmic improvements have made it possible to solve nearly all standard instances from the literature [Pisinger]
    - 0/1 variant introduced in 1980 by Gallo, Hammer and Simeone [Billionnet]

## Importance

* + - “used in computer science to explore how algorithms work and how efficient they are.” [Hurbans]
    - Real World Uses
      * *The versatility of the Knapsack is evident in many real-world applications, from efficiently packing food for survival [Rocca] to optimizing investment portfolios, loading cargo planes, and even cutting logs [Kellerer]. This ability to maximize efficiency and profitability has made it a go-to solution in a multitude of diverse industries.*
      * When packing food for survival [Rocca]
        + Weight: The weight of each food item to be carried.
        + Value: The nutritional value/ caloric content of each food item
      * Investment Portfolios [Kellerer]
        + Weight: The amount of money required for each investment
        + Value: The expected net return or profit for each investment
      * Loading Cargo Planes [Kellerer]
        + Weight: The weight of each package
        + Value: The profit earned by the cargo company for each package
      * Cutting Logs
        + Weight: The predefined lengths into which the log can be cut.
        + Value: The selling price associated with each cut piece.
        + *While often considered a ‘packing’ problem, it can also be viewed as a ‘cutting problem’*
      * Program Partitioning/Task Allocation
        + Weight: computational resources/ time required to execute task/ program. (Tasks or programs with higher computational requirements would have higher weights.)
        + Value: importance/priority of the task or program. (Tasks or programs with higher priority or value would have higher values.)
    - Segue
      * *Unfortunately, the real world often presents additional complexities that need to be considered when calculating the best value. 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. (Kellerer)*

## Variants

### Introduction/ 0/1 knapsack problem

* + - * 0/1 formula
        + The 0/1 Knapsack Problem is a combinatorial optimization problem where each item can either be included entirely (0) or not included at all (1) in the knapsack, making it a discrete and more restrictive version compared to the Fractional Knapsack Problem which allows fractions of items to be included in the knapsack.[KPA]

### Greedy

* + - * “a greedy algorithm is an algorithm that doesn’t try multiple options: it tries just one.” [Zingaro]
      * makes locally optimal choices at each step without considering the overall consequences [KPA]
      * naïve and heuristic methods
      * “intuitive” [Kellerer]
      * selects items based on their value-to-weight ratio, sorts them in descending order by ratio, and fills the bag until it reaches as close to max weight as possible
      * Efficient and Simple to understand and implement
        + “run faster and are easier to implement than dynamic-programming algorithms” [Zingaro]
      * Does not always produce the most optimal solution
        + Due to its efficient and approximate solutions, it should only be used in scenarios where "good enough" solutions are acceptable

Returning change using the least amount of coins

A variant of the shortest path problem - For example, Dijkstra's algorithm, uses a Greedy approach [Zingaro]

* + - * + “not as broadly applicable as other algorithm design approaches (such as dynamic programming).” [Zingaro]
        + “when they do happen to work, it’s often for subtle, problem-specific reasons” [Zingaro]
      * *As such, in many cases Dynamic Programming is a better approach…*

### Dynamic Programming

#### Introduction

* + - * + Solves problems by breaking them down into smaller subproblems and solving those [Bhargava]

“Start by solving only a small subproblem of (KP) and then extend this solution iteratively until the complete problem is solved.” [Kellerer]

Instead of dealing with all n problems at once, iteratively add them to the problem [Kellerer]

This is done by using the Bellman Recursion[Kellerer]

used to compute the optimal solution values for the knapsack subproblems by iteratively adding items to the knapsack and considering all possible choices to maximize the total value.[KPA]

* + - * + “Dynamic programming is one of our best approaches for solving difficult (KP), since this is the only solution method which gives us a worst-case guarantee on the running time, independently on whether the upper bounding tests will work well” Study by Pisinger
        + DP works in two phases: in the forward phase, we calculate the optimal value of the profit function by tabulation of a recurrence equation, and then we use this in the backtracking phase to determine an actual solution which has this optimal profit.[]

#### Knapsack [Bhargava]

* + - * + Break the knapsack into smaller knapsacks, solve those, and work up towards the original one
        + To do this, the algorithms starts with a grid with items as the rows and columns as weights

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Vase |  |  |  |  |
| Watch |  |  |  |  |
| Artifact |  |  |  |  |

Each column represents a smaller knapsack

Since our smallest item weighs 2kg, we don’t include column 1

* + - * + The table is filled out, row by row, asking at each cell “will stealing this item give me the most value”?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | w | w | w | w |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg |  |  |  |  |
|  |  |  |  |
| Artifact 100k, 4kg |  |  |  |  |
|  |  |  |  |

Since our only choice in row 1 is the watch, and it fits in all our smaller bags, it is chosen for each cell

* + - * + Once you continue to the next row, you can either steal the item at that row or from any of the rows above it

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | w | w | w | w |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | w |  |  |  |
| 80k |  |  |  |
| Artifact 100k, 4kg |  |  |  |  |
|  |  |  |  |

The vase is too heavy for the 2 bag, so the watch has to be chosen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | w | w | w | w |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | w | v | v |  |
| 80k | 90k | 90k |  |
| Artifact 100k, 4kg |  |  |  |  |
|  |  |  |  |

But for the 3 and 4 bags, the vase is worth 90k while the watch is only worth 80k. So the vase is the obvious choice

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | w | w | w | w |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | w | v | v | vw |
| 80k | 90k | 90k | 170k |
| Artifact 100k, 4kg |  |  |  |  |
|  |  |  |  |

For bag 5, while the vase seems like the best choice at first – notice we would have 2kg left over in space.

All we have to do is look at the 2 column to find the best use of that space – the watch. Then we just add their values together

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | w | w | w | w |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | w | v | v | vw |
| 80k | 90k | 90k | 170k |
| Artifact 100k, 4kg | w | v |  |  |
| 80k | 90k |  |  |

Artifact doesn’t fit in bags 2 and 3 so we just bring down the cells from above

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | W | W | W | W |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | W | V | V | VW |
| 80k | 90k | 90k | 170k |
| Artifact 100k, 4kg | W | V | A |  |
| 80k | 90k | 100k |  |

For bag 4, the artifact is the better choice because 100k>90k

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 2 | 3 | 4 | 5 |
| Watch 80k, 2kg | W | W | W | W |
| 80k | 80k | 80k | 80k |
| Vase 90k, 3kg | W | V | V | VW |
| 80k | 90k | 90k | 170k |
| Artifact 100k, 4kg | W | V | A | VW |
| 80k | 90k | 100k | 170k |

For bag 5, we just bring down the cell above because 170k>100k

Which gives us our answer

#### Formula:

* + - * + Where and

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Max value that can be achieved with items and capacity | |
|  | | The value from the cell just above the current cell, (represents the maximum value achieved without considering the current, th, item) | |
|  | | value of the current, th, item | |
|  | | The value from the previous row, 'weight' positions to the left (column 'j - weight').  (represents the best value possible for the remaining weight in the knapsack after item is included) | |
|  | | compares and returns the larger value of items and | |
|  |  | The maximum value achieved without considering the current, th, item  Stored in the cell above the current cell |
|  |  | The value of item plus the best value achievable for the remaining capacity after taking the current item's weight |

# Algorithms:

## Instructions:

* + - “You must present pseudocode for at least two different algorithms in your report that solve the knapsack problem. The algorithms must be from different paradigms (dynamic programming, backtracking, naive Divide and Conquer, etc.). New and obscure implementations are welcomed, but not necessary.”

## Introduction

* + - Describe Brute Force briefly and why it isn’t a good option
    - Explain IO

## Algorithm 1: Naïve – Extended Greedy

### Special Case [Kellerer]

* + - * Consider the following values:
        + n = 2 (only two items)
        + c = 3 (knapsack capacity)
        + Item 1: weight (w\_1) = 1, profit (p\_1) = 2, efficiency (e\_1) = 2
        + Item 2: weight (w\_2) = 3, profit (p\_2) = 3, efficiency (e\_2) = 1
      * In this example, the Greedy Algorithm starts by selecting Item 1, as it has a higher efficiency (e\_1 = 2) compared to Item 2 (e\_2 = 1). The algorithm packs Item 1 into the knapsack, leaving a remaining capacity of 2. Since no additional items can fit within this space, it considers the knapsack full and the function exits. However, the optimal solution, in this case, would be to pack Item 2 with a weight of 3.
      * As such, we will implement an extended greedy algorithm which will incorporate 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.

### Pseudo code

* + - * Initialize total weight and profit variables to 0
      * Sort items by efficiency in decreasing order (highest first)
      * iterate sorted items, For each item:
        + If adding the item does not violate exceed weight capacity

include the item

Update total weight

Update total profit

* + - * + Else

skip adding the item to the knapsack

## Algorithm 2: Dynamic Programming [1/0]

### Bellman Recursion

If the knapsack capacity d is less than the weight of item j (i.e., d < w\_j), then the optimal solution value z\_j(d) for the subproblem with item set {1, ..., j} and capacity d remains unchanged from the solution value of the subproblem with item set {1, ..., j-1} and the same capacity: z\_j(d) = z\_(j-1)(d).

If the knapsack capacity d is greater than or equal to the weight of item j (i.e., d >= w\_j), then the optimal solution value z\_j(d) for the subproblem with item set {1, ..., j} and capacity d is computed as the maximum of two choices:

a) If item j is not packed into the knapsack, the solution value remains unchanged: z\_j(d) = z\_(j-1)(d).

b) If item j is packed into the knapsack, it contributes p\_j to the solution value, and the remaining capacity is reduced to d - w\_j. The best possible solution value for this reduced capacity is given by z\_(j-1)(d - w\_j). Therefore, in this case, z\_j(d) = max(z\_(j-1)(d), z\_(j-1)(d - w\_j) + p\_j).

The Bellman recursion is applied iteratively for each item j and each capacity d from 0 to the total knapsack capacity c. By following this recursion, the algorithm fills up a dynamic programming table DP, which contains the optimal solution values for different subproblems. Ultimately, the value at DP[n][c] gives the overall optimal solution value for the original knapsack problem (KP).

The recursion considers the trade-off between including or excluding each item in the knapsack, and by taking the maximum value, it ensures that the final solution is the one that maximizes the total value while respecting the knapsack capacity constraint. The algorithm efficiently solves not only the single-capacity knapsack problem but also the all-capacities knapsack problem, providing the optimal solution for each possible capacity from 0 to c. The resulting algorithm is referred to as DP-1, or dynamic programming by weights.

### Psuedo Code

## Misc

* + - “Performance is defined as how well a specific solution solves a problem, not necessarily computational performance.” [Hurbans]

# Time Complexity:

## Instructions:

* + - “For each algorithm presented above, cite your sources and you may use their justification for the time-complexity. Feel free to utilize online resources to help in this, but be comfortable enough that if you were asked questions about the work, you could answer them. Huge leaps in logic or math will likely be met with questions. YOU DO NOT NEED TO PROVE ALGORITHM CORRECTNESS OR TIME COMPLEXITY. We will assume correctness for now.”

## Algorithm 1:

* + - a

## Algorithm 2:

* + - A

# MISC

* + Is an NP-Hard Problem
    - “can be solved in pseudo-polynomial time through dynamic programming” [Pisinger]

# Code:

## Instructions:

* + - “Write up the algorithms you presented above in the coding language of your choice. Using similar input, note the difference in their respective observed runtimes in your report and why there might be deviations from our expectations of the time complexity given above. Provide screenshots and instructions on how to run your programs in your report. Also, submit these code files along with your report.”

## A

* + - a

# Sources [Citations]:

## 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

* + - <https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/>

## 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