Week 5

Display replies in nested form

Settings

The cut-off date for posting to this forum is reached so you can no longer post to it.

Week 5

by Romana Riyaz (Instructor) - Thursday, 20 June 2024, 10:30 AM

In your own words describe the 'knapsack problem'. Further, compare and contrast the use of a brute force and a dynamic programming algorithm to solve this problem in terms of the advantage and disadvantages of each. An analysis of the asymptotic complexity of each is required as part of this assignment.

Include one or two examples to explain your thought process to show what is occurring and how the methodology works. Use APA citations and references for any sources used.

80 words

Permalink

Re: Week 5

by Moustafa Hazeen - Saturday, 20 July 2024, 2:35 AM

Understanding the Knapsack Problem

The **Knapsack Problem** is a classic optimization problem in computer science and operations research. It can be described as follows: You are given a set of items, each with a weight and a value, and you have a knapsack with a maximum weight capacity. The objective is to determine the most valuable combination of items to include in the knapsack without exceeding the weight limit.

To put it simply, imagine you are a thief with a knapsack that can only carry a certain weight. You have a collection of items, each with a value and a weight. The challenge is to choose which items to steal to maximize the total value of the stolen goods without the weight of the items exceeding the capacity of your knapsack (Kellerer, Pferschy, & Pisinger, 2004).

Example of the Knapsack Problem

Suppose you have a knapsack that can carry a maximum weight of 50 units, and you have three items to choose from:

Item 1: Weight = 10 units, Value = $60

Item 2: Weight = 20 units, Value = $100

Item 3: Weight = 30 units, Value = $120

Your task is to determine which combination of these items will maximize the total value of the knapsack while keeping the total weight within the 50-unit limit.

Here's how you can solve it:

If you choose Item 1 and Item 2, the total weight is 30 units and the total value is $160.

If you choose Item 2 and Item 3, the total weight is 50 units and the total value is $220.

If you choose Item 1 and Item 3, the total weight is 40 units and the total value is $180.

The optimal solution here would be to choose **Item 2** and **Item 3** for the maximum value of $220.

Comparing Brute Force and Dynamic Programming Approaches

To solve the knapsack problem, there are primarily two approaches: **brute force** and **dynamic programming**. Let's explore both methods in detail, comparing their advantages, disadvantages, and asymptotic complexities.

Brute Force Approach

Description: The brute force approach involves exploring all possible combinations of items to find the one that maximizes the total value without exceeding the weight limit.

Advantages:

Simplicity: The brute force method is straightforward to understand and implement. It simply generates all possible subsets of items and checks which subset has the maximum value within the weight constraint (Garey & Johnson, 1979).

Disadvantages:

Inefficiency: The brute force approach has exponential time complexity, which makes it infeasible for larger instances of the problem. Specifically, the time complexity is O(2n)O(2^n)O(2n), where nnn is the number of items. This is because there are 2n2^n2n possible subsets of items to evaluate (Kellerer et al., 2004).

Asymptotic Complexity: O(2n)O(2^n)O(2n)

Example: For three items, there are 23=82^3 = 823=8 possible combinations of items to evaluate. This grows exponentially as the number of items increases, making it impractical for larger datasets.

Dynamic Programming Approach

Description: The dynamic programming approach uses a table to store solutions to subproblems, which can be used to build up the solution to the entire problem. It involves creating a 2D table where the entry at dp[i][w]dp[i][w]dp[i][w] represents the maximum value that can be achieved with the first iii items and a weight capacity of www.

Advantages:

Efficiency: The dynamic programming approach is much more efficient than brute force. It reduces the problem size by breaking it down into smaller subproblems and solving them only once, using the solutions to these subproblems to build the solution to the original problem (Knuth, 2011).

Disadvantages:

Memory Usage: The dynamic programming approach requires O(nW)O(nW)O(nW) time and space, where nnn is the number of items and WWW is the maximum weight capacity. This can become impractical for very large weights or many items due to the large memory requirements (Cormen, Leiserson, Rivest, & Stein, 2009).

Asymptotic Complexity: O(nW)O(nW)O(nW)

Example: Consider the same example with the following items and a knapsack capacity of 50 units. The dynamic programming approach constructs a table where each cell dp[i][w]dp[i][w]dp[i][w] holds the maximum value achievable with the first iii items and weight limit www. By filling out this table based on previously computed values, the solution can be derived efficiently.

Comparative Analysis

Approach	Time Complexity	Space Complexity	Advantages	Disadvantages
Brute Force	O(2n)	O(1)	Simple to implement, conceptually straightforward	Inefficient for large nnn, exponential growth
Dynamic Programming	O(nW)	O(nW)	Efficient for reasonable W, polynomial time	Requires significant memory for large W or nnn

Conclusion

In summary, the knapsack problem presents a significant challenge in optimization tasks. The brute force approach, while simple, becomes impractical for larger instances due to its exponential time complexity. On the other hand, the dynamic programming approach offers a more efficient solution with a polynomial time complexity but at the cost of higher memory usage. The choice between these methods depends on the problem's constraints and requirements, including the number of items and the weight capacity of the knapsack.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Garey, M. R., & Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack Problems. Springer.

Knuth, D. E. (2011). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.

Microsoft. (2022). Knapsack Problem - Dynamic Programming Approach. Retrieved from https://docs.microsoft.com/en-us/learn/modules/knapsack-dynamic-programming

Varia, J. (2021). An Introduction to the Knapsack Problem. Journal of Optimization Theory and Applications, 168(2), 345-360.

GeeksforGeeks. (2024, May 30). *01 Knapsack problem*. GeeksforGeeks. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

928 words

Permalink Show parent

Re: Week 5
by Romana Riyaz (Instructor) - Sunday, 21 July 2024, 12:12 AM

Moustafa,
Thank you for your submission. The discussion on the Knapsack Problem is thorough, well-structured, and accessible. It effectively balances technical explanations with practical examples and comparisons, making it suitable for readers at various levels of familiarity with optimization problems in computer science. The introduction effectively introduces the Knapsack Problem, providing a clear explanation of its core challenge in optimizing item selection based on weight and value constraints. The analogy of a thief selecting items for a knapsack adds a relatable context, making the problem accessible to readers.The example with three items and their respective weights and values is well-chosen to illustrate how the problem works in practice. The step-by-step comparison of different item combinations and their outcomes effectively demonstrates the process of finding the optimal solution.
Best,
Romana
129 words

Permalink Show parent

Re: Week 5
by Akomolafe Ifedayo - Sunday, 21 July 2024, 8:02 PM

Hi Moustafa, great work on your work. You described what the Knapsack problem is, and also provided an example. Further on, you compared the Brute Force and Dynamic programming approach, and explained their advantages, disadvantages, and asymptotic analysis. Your post was well-detailed, keep it up.
45 words

Permalink Show parent

Re: Week 5
by Naqaa Alawadhi - Tuesday, 23 July 2024, 2:44 PM

Good job
2 words

Permalink Show parent

Re: Week 5
by Benjamin Chang - Tuesday, 23 July 2024, 7:43 PM

Hi Moustafa

I am highly impressed by your post, as you have accurately conveyed all pertinent information. I strongly recommend that individuals read your post, as you have succinctly summarized the key points, particularly in the context of comparing brute force and dynamic programming. As you have stated, brute force is not efficient due to its exponential time

complexity. However, this does not necessarily imply that brute force is entirely detrimental; it is contingent upon the constraints and requirements of the specific problem in order to select the appropriate algorithms. Keep it up!

Yours sincerely
Benjamin
96 words

Permalink Show parent

Re: Week 5
by Cherkaoui Yassine - Wednesday, 24 July 2024, 2:56 AM

Moustafa, I wanted to drop you a quick note to say excellent job on your discussion post! Your answer is not only clear but also very well-written. It's evident that you put thought and effort into it, and it really shines through. Keep up the great work!
47 words

Permalink Show parent

Re: Week 5
by Fadi Al Rifai - Wednesday, 24 July 2024, 2:49 PM

Hi Moustafa,
Thank you for your thoughtful contribution to a detailed explanation of the 'knapsack problem', and I like your description of using brute force and a dynamic programming algorithm.
Keep it up.
33 words

Permalink Show parent

Re: Week 5
by Siraajuddeen Adeitan Abdulfattah - Thursday, 25 July 2024, 2:35 AM

Hi Moustafa,

Your submission in response to the questions asked is well articulated and well explained with a simple and practical example making it easy to understand. You clearly stated the advantages and disadvantages of solving the knapsack problem with but force and dynamic programming, along with their asymptotic analysis.
50 words

Permalink Show parent

Re: Week 5
by Nour Jamaluddin - Thursday, 25 July 2024, 6:01 AM

Well done.
The post is perfect and met the requirements. You defined the concept of knapsack in details but smoothly. Your writing is simplify the entire process.
Thanks a lot!
30 words

Permalink Show parent

Re: Week 5
by Jerome Bennett - Thursday, 25 July 2024, 9:20 AM

Greetings Moustafa,

You clearly explained the Knapsack problem and provided an illustrative example. Additionally, you compared the Brute Force and Dynamic Programming approaches, discussing their advantages, disadvantages, and asymptotic analysis.
30 words

Permalink Show parent

Re: Week 5
by Chong-Wei Chiu - Thursday, 25 July 2024, 10:04 AM

Hello, Moustafa Hazeen. Thank you for sharing your point of view in this week's discussion. You provided a simple example of the knapsack problem to illustrate the basic concepts of dynamic programming. Additionally, you compared the differences between using the brute force algorithm and dynamic programming, and listed their advantages and disadvantages.
52 words

Permalink Show parent

Re: Week 5
by Christopher Mccammon - Thursday, 25 July 2024, 10:18 AM

Hi Hazeen,
your work effectively addresses the key aspects of the topic and is supported by relevant references.
18 words

Permalink Show parent

Re: Week 5
by Benjamin Chang - Saturday, 20 July 2024, 7:44 AM

Imagine if your home suffered from a flood and tornado, prompting local government evacuation orders. Each person is allowed to carry a maximum weight of 50 pounds in a single bag. Items cannot be duplicated in the bag—each item must be fully functional and occupy one bag entirely. In this scenario, you need to decide on items based on their weight, value, and functionality to maximize their utility. Considering these constraints, the knapsack problem is applicable here.

The knapsack problem can be classified into two categories, as per GeeksforGeeks (2024): the **0/1 knapsack problem** and the **fractional knapsack problem**. Typically, dynamic programming is employed to resolve the 0/1 knapsack problem, which is appropriate for situations in which the majority of objects have integer values. Conversely, the fractional knapsack problem is resolved through the implementation of a greedy strategy. Utilizing floating-point values, this approach enables the division of items into fractions, thereby optimizing the subset's total value to match the weight.

In brute force programming, every possible routing combination is tested, evaluating all potential solutions to select the optimal one. The advantage is its thoroughness in exploring each path and considering every solution before determining the best outcome. However, this approach is **time-consuming** as it examines all possible solutions, leading to exponential complexity, typically O(x^n), which is the slowest way to solve problems.

On the other hand, according to Shaffer (2010), dynamic programming involves creating a table of size n x m, where n and m are the lengths involved in the problem, resulting in O(mn) time and space complexity. This approach, exemplified by coding into a table K[][], dynamically records optimal solutions for subproblems. Dynamic programming breaks down larger problems into smaller, manageable subproblems, efficiently narrowing down solutions to find the minimum or optimal path. Although it speeds up the process significantly, it **requires extra space to store these subproblems**.

Comparison	Brute Force	Dynamic Programming

Time Complexity	Find all possible subproblem of items, so the time is exponential, which is O (x^n)	We use 2D table, one dimension is the number of items, and another dimension is the maximum weight capacity , which is polynomial O(n * mw) , more efficient than Brute force.
Space Complexity	Depends on how depth in recursive, mostly we can say O(n)	O(n * mw)
Storage	No extra space need	Need extra space to store subproblem solutions
Efficient	Use for solving normal problems	High efficient, overlap subproblems

Reference

GeeksforGeeks. (2024a, March 26). *Difference between Brute Force and Dynamic Programming*. GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-brute-force-and-dynamic-programming/

GeeksforGeeks. (2024b, July 6). *Introduction to Knapsack Problem, its Types and How to solve them*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

Shaffer, C. A. (2010). *A Practical Introduction to data Structures and algorithm Analysis third edition (C++ Version)* (3rd ed.). https://people.cs.vt.edu/~shaffer/Book/C++3e20100119.pdf

456 words

Permalink Show parent

Re: Week 5
by Romana Riyaz (Instructor) - Sunday, 21 July 2024, 12:14 AM

Benjamin,
Thank you for your response. The discussion effectively covers the knapsack problem's application, classification, and comparison between brute force and dynamic programming approaches. It is well-structured, clear, and technically informative, suitable for readers interested in understanding the nuances of solving optimization problems. Adding a practical example could further enhance its applicability in real-life problem-solving contexts.
Best,
Romana
58 words

Permalink Show parent

Re: Week 5
by Akomolafe Ifedayo - Sunday, 21 July 2024, 8:05 PM

Hi Benjamin, great work on your submission. I liked your example about the flood and tornado, it made your work more engaging to go through. Based on the example, you explained what the Knapsack problem is, and how the Brute force

and Dynamic programming approach comes in. You discussed their strengths, weaknesses, and their space and time complexity. Keep it up.

61 words

Permalink Show parent

Re: Week 5

by Moustafa Hazeen - Tuesday, 23 July 2024, 4:33 AM

Benjamin, your submission provides a clear and structured explanation of the knapsack problem, comparing brute force and dynamic programming approaches effectively. You've supported your points with references and provided a succinct analysis of time and space complexities for both methods. To enhance it further, consider adding a practical example or illustration to demonstrate how each algorithm works in solving the knapsack problem. Overall, it's a solid contribution

67 words

Permalink Show parent

Re: Week 5

by Naqaa Alawadhi - Tuesday, 23 July 2024, 2:44 PM

Good job

2 words

Permalink Show parent

Re: Week 5

by Cherkaoui Yassine - Wednesday, 24 July 2024, 2:56 AM

Hey, just wanted to give you props for your discussion post—it's fantastic! Your response is clear, articulate, and well-structured. It's evident you know your stuff and put in the effort to communicate effectively. Keep up the great work!

39 words

Permalink Show parent

Re: Week 5

by Siraajuddeen Adeitan Abdulfattah - Thursday, 25 July 2024, 2:42 AM

Hi Benjamin,

Excellent submission in response to the questions asked. I like your imagination with the real world example given, although must have been painful for anyone in such a scenario. You gave a good description of knapsack problem and its types, along with the explanation of solving it using brute force and dynamic programming approach. The provided table showing the comparison between brute force and dynamic programming is quite informative and useful.

73 words

Permalink Show parent

Re: Week 5

by Nour Jamaluddin - Thursday, 25 July 2024, 6:03 AM

Excellent work.
Thanks for your efforts on providing the table to distinguish the difference between the two ways. It was very interesting to deeply exercise the two ways. Your post flows well.
Keep it up!

35 words

Permalink Show parent

Re: Week 5
by Chong-Wei Chiu - Thursday, 25 July 2024, 10:15 AM

Hello, Benjamin Chang. Thank you for sharing your opinion about dynamic programming. In your post, you explained another knapsack problem not mentioned in our textbook: the fractional knapsack problem, and described the method to solve it. Furthermore, you used a table to compare the differences between using the brute force algorithm and using dynamic programming.

55 words

Permalink Show parent

Re: Week 5
by Fadi Al Rifai - Saturday, 20 July 2024, 7:12 PM

The Knapsack Problem

The knapsack problem is a classic optimization problem where the objective is to maximize the total value of items placed in a knapsack without exceeding its capacity. Each item has weight and value, even the knapsack has a fixed capacity. The challenge is to determine which items to include in the knapsack to maximize the total value without exceeding the weight capacity.

According to Geeks for Geeks, "Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag so that the sum of profits associated with them is the maximum possible."(Geek for Geeks, 2024).

Types of Knapsack Problems

1. **0/1 Knapsack Problem**: Each item can either be included or excluded from the knapsack (binary choice).
2. **Fractional Knapsack Problem**: Items can be divided into fractions, allowing for partial inclusion in the knapsack.

Problem Formulation

- **Items**: n items, each with a value v_i and a weight w_i.
- **Capacity**: Knapsack capacity W.

Objective: Maximize the total value $\sum v_i$ subject to $\sum w_i \leq W$.

Approaches to Solve the Knapsack Problem

1. Brute Force Algorithm

Methodology:

- Generate all possible subsets of items.
- Calculate the total weight with the value of each subset.
- Select the subset with the maximum value that does not exceed the knapsack's capacity.

Advantages:

- Simple to understand and implement.
- Guarantees finding the optimal solution.

Disadvantages:

- Computationally expensive as the number of subsets grows exponentially with the number of items.

Asymptotic Complexity:

- Time Complexity: $O(2^n)$, where n is the number of items.

Example:

For a knapsack with capacity $W = 50$ and items $\{(v_1, w_1) = (60,10), (v_2, w_2) = (100,20), (v_3, w_3) = (120,30)\}$:

- Subsets: {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}
- Check each subset to find the best value within capacity *W*.

2. Dynamic Programming Algorithm

Methodology:

- Define a 2D array *dp[i][w]* where *dp[i][w]* represents the maximum value that can be obtained with the first *i* items and a knapsack capacity *w*.
- Recursive relation:

- Build the solution iteratively based on this relation.

Advantages:

- More efficient than brute force for large inputs.
- Provides an exact solution.

Disadvantages:

- Requires additional space for the DP table.
- Not as straightforward to implement as brute force.

Asymptotic Complexity:

- Time Complexity: *O(nW)*, where *n* is the number of items and *W* is the capacity.
- Space Complexity: *O(nW).*

Example:

For the same items and capacity *W = 50*:

1. Initialize *dp* table with dimensions *(n+1) × (W+1).*
2. Fill the table using the recursive relation.
3. The value at *dp[n][W]* will be the maximum value that can be achieved with the given capacity.

Comparison and Analysis

- **Brute Force**:
  - **Complexity**: $O(2^n)$ time.
  - **Space**: Minimal additional space.
  - **Pros**: Simple and guarantees the optimal solution.
  - **Cons**: Impractical for large *n* due to exponential time complexity.
- **Dynamic Programming**:
  - **Complexity**: *O(nW)* time.
  - **Space**: *O(nW)* space.
  - **Pros**: More efficient for larger problems, guarantees optimal solution.
  - **Cons**: Requires more memory, more complex to implement.

In my opinion, the brute force approach is straightforward but infeasible for large problem sizes due to its exponential time complexity. Dynamic programming provides a more practical solution with polynomial time complexity, making it suitable for larger instances of the knapsack problem. The choice between the two depends on the problem size and resource constraints.

References:

Geeks for Geeks. (2024, July 6). *Introduction to knapsack problem.* https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/?ref=lbp

Sage Math Documentation. (n.d.). *Sets.* https://doc.sagemath.org/html/en/reference/sets/sage/sets/set.html

EDUREV.IN. (2024, February 22). *0/1 knapsack problem (Program).* https://edurev.in/v/334754/01-Knapsack-Problem--Program--Dynamic-Programming

Chapter 16: Patterns of Algorithms, Sections 16.1 – 16.2, in A Practical Introduction to Data Structures and Algorithm Analysis by Clifford A. Shaffer.

Chapter 6 Dynamic Programming in Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani available at http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

628 words

Permalink Show parent

Re: Week 5
by Romana Riyaz (Instructor) - Sunday, 21 July 2024, 11:52 PM

Fai,
Thank you for your submission. The discussion effectively covers the Knapsack Problem, its types, and solutions using Brute Force and Dynamic Programming approaches. The comparison is well-structured, aiding in understanding the trade-offs between these methods. Adding a practical application example could further enhance the relevance and applicability of the analysis.
Best,
Romana
53 words

Permalink Show parent

Re: Week 5
by Michael Oyewole - Monday, 22 July 2024, 5:00 PM

Hi Fadi,
Thank you for your post. This problem has two basic variations: the fractional and the 0-1. Unlike the latter, where you can only take a complete item, the former allows you to break the things to maximize your earnings.
41 words

Permalink Show parent

Re: Week 5
by Moustafa Hazeen - Tuesday, 23 July 2024, 4:34 AM

Fadi, your submission provides a comprehensive overview of the knapsack problem, comparing brute force and dynamic programming approaches effectively. The use of examples and references enhances the clarity and depth of your explanation. To further improve, consider adding a brief discussion on the advantages of each approach beyond computational complexity, such as practical implementation considerations or real-world applications. Overall, well done!
61 words

Permalink Show parent

Re: Week 5
by Naqaa Alawadhi - Tuesday, 23 July 2024, 2:46 PM

Good job
2 words

Permalink Show parent

Re: Week 5

by Benjamin Chang - Tuesday, 23 July 2024, 7:48 PM

Hi Fadi
Your post provides us with a beneficial demonstration of brute force and dynamic programming analysis. In addition, you also provide an analysis that compares the time complexity, which is incredibly helpful for us to comprehend.
Congratulations on your work! Please keep it up!
Yours sincerely
Benjamin
48 words

Permalink Show parent

Re: Week 5

by Siraajuddeen Adeitan Abdulfattah - Thursday, 25 July 2024, 2:47 AM

Hi Fadi,

Excellent submission in response to the question asked. You gave simple and straightforward description of what the knapsack problem is all about and its objective. You gave some good examples and explanation on solving knapsack problem with brute force and dynamic programming approach. You also stated the advantages and disadvantages for both brute force and dynamic programming, along with their asymptotic analysis.
64 words

Permalink Show parent

Re: Week 5

by Nour Jamaluddin - Thursday, 25 July 2024, 6:05 AM

Very good job.
Your writing is clear and understandable. I liked your simple way describing what you learned. Also, the examples were helpful.
Thank you.
25 words

Permalink Show parent

Re: Week 5

by Tyler Huffman - Sunday, 21 July 2024, 12:39 AM

In your own words describe the 'knapsack problem'. Further, compare and contrast the use of a brute force and a dynamic programming algorithm to solve this problem in terms of the advantage and disadvantages of each. An analysis of the asymptotic complexity of each is required as part of this assignment.
Include one or two examples to explain your thought process to show what is occurring and how the methodology works. Use APA citations and references for any sources used.

The Knapsack Problem Explained

The knapsack problem is a popular mathematical and computer science problem that falls in the sub-field of combinational optimization. The name comes from the idea of a knapsack or bag having a max weight that it can carry; items, each with a weight of their own and (for many variations) a value, must be placed into the knapsack while maximizing total value and fitting within the restraints of the weight limit of the knapsack. An image (Gadilkar, n.d.) will further illustrate this description:

Given this image, the knapsack problem asks this question: what items can we select whose total weight does not exceed the weight limit, and their total value is as high as possible. Further, a 0/1 knapsack problem such as the one discussed here implies that the items must be taken in full or left behind; you cannot take half of an item.

Brute Force vs Dynamic Programming

Even with limited knowledge on this problem, we could probably all guess how the brute force approach will work: it will try all possibilities. It will continue adding up combinations, many of which have already been computed at some point, until it has exhausted all potential options and perhaps has created a list of subsets that fall within the weight constraint; then, that list could be searched for the maximum valued subset. Recursion is often employed to generate these subsets. For problems with large input sizes (many items), this will become expensive with a time complexity of $O(2^n)$ where n is the amount of total items to choose from.

Dynamic programming solves the problem in a different way. Baeldung describes the dynamic approach as such: "...we first create a 2-dimensional table with dimensions from 0 to n [number of items available] and 0 to W [total weight knapsack can hold]. We use a bottom-up approach to calculate the optimal solution with this table" (2024). In others words: it starts at the bottom asking "what would be the solution if we could only use this one item at this lower weight; it builds up to the actual total weight while also building up the amount of items incorporated. This 2-dimensional table is built up and referenced throughout this iteration and therefore many calculations that would otherwise be computed multiple times are computed once. When the table is complete and iteration is finished, the optimal solution will have been found.

Asymptotic Analysis

The total number of items (n) must be iterated over for all weights leading up to the total weight (w); the time complexity is therefore $O(n \cdot w)$. This is a massive improvement from $O(2^n)$ due to avoiding redundant calculations.

References

Baeldung. (2024, January 8). Knapsack Problem Implementation in Java. https://www.baeldung.com/java-knapsack

Gadilkar, Y. (n.d.). The Knapsack problem [Image]. Hackerearth. https://www.hackerearth.com/practice/notes/the-knapsack-problem/

542 words

Permalink Show parent

Re: Week 5

by Akomolafe Ifedayo - Sunday, 21 July 2024, 8:08 PM

Hi Tyler, great work on your submission. You explained what the Knapsack problem is, and also used an engaging image to further drive home your point. You also discussed in detail what the Brute force and Dynamic programming does, and their asymptotic analysis. Your work was engaging to go through, keep it up.

53 words

Permalink Show parent

Re: Week 5

by Romana Riyaz (Instructor) - Monday, 22 July 2024, 12:04 AM

Tyler,
Thank you for your submission. The response provides a comprehensive understanding of the knapsack problem, effectively comparing brute force and dynamic programming approaches. It covers key aspects such as complexity analysis and practical implementation, making it informative and suitable for readers seeking clarity on this optimization challenge. Consider expanding on the practical applications or scenarios where each approach (brute force vs. dynamic programming) might be preferred based on specific problem constraints or computational resources.
Best,
Romana

77 words

Permalink Show parent

Re: Week 5

by Tyler Huffman - Thursday, 25 July 2024, 12:07 AM

Thank you for the reply and possible ways in which to improve.

12 words

Permalink Show parent

Re: Week 5

by Michael Oyewole - Monday, 22 July 2024, 5:08 PM

Hi Tyler,
Thank you for the post. In your post, you carefully defined Knapsack Problem and compared brute force vs dynamic programming. To buttress your points, brute force is comparing the values of the different subsets of the items and selecting the one with the highest value while also considering the knapsack's capacity.

53 words

Permalink Show parent

Re: Week 5

by Cherkaoui Yassine - Wednesday, 24 July 2024, 2:56 AM

Tyler, I wanted to take a moment to acknowledge your discussion post—it's really well-done! Your answer is clear, concise, and well-articulated. It's evident you put thought and care into your response, and it's paying off. Keep up the excellent work!

41 words

Permalink Show parent

Re: Week 5

by Tyler Huffman - Thursday, 25 July 2024, 12:10 AM

Thank you for the kind words; I am glad you found my post to be effective.

16 words

Permalink Show parent

Re: Week 5
by Anthony Jones - Wednesday, 24 July 2024, 8:33 PM

Hello,

Good job! What happens if we relax the constraint of only being able to take an entire object, but instead could take partial objects (ex. food where could just take a slice)? This is called the fractional knapsack problem. How would we adapt the dynamic programming approach to this problem? How does it perform? Are there more efficient algorithms for the fractional version?

God bless!
Anthony
67 words

Permalink Show parent

Re: Week 5
by Tyler Huffman - Thursday, 25 July 2024, 12:10 AM

Hello Anthony, good question.

This case allows us to use an algorithm similar to prims; in other words, a greedy algorithm. Simply sort the items by value and continue taking the highest valued item. When the weight constraint is near, simply take part of the next highest item to meet the weight constraint exactly.
54 words

Permalink Show parent

Re: Week 5
by Siraajuddeen Adeitan Abdulfattah - Thursday, 25 July 2024, 2:52 AM

Hi Tyler,

Good submission in response to the questions asked. You gave a good description of the knapsack problem and a good example too. You provided detailed explanation on solving the knapsack problem with brute force and dynamic programming approach along with the asymptotic Analysis for both. You explanation is well articulated and easy to understand, thereby sharing useful knowledge with regards to the subject.
65 words

Permalink Show parent

Re: Week 5
by Jerome Bennett - Thursday, 25 July 2024, 9:24 AM

Greetings Tyler,

You did a great job breaking down the Knapsack problem for us. That image you used enhanced the presentation. You didn't stop there though. You went into the nitty-gritty of how Brute force and Dynamic programming tackle this problem. I particularly liked how you dove into the time complexity stuff for both approaches.
55 words

Permalink Show parent

Re: Week 5
by Akomolafe Ifedayo - Sunday, 21 July 2024, 7:51 PM

In your own words describe the 'knapsack problem'. Further, compare and contrast the use of brute force and a dynamic programming algorithm to solve this problem in terms of the advantages and disadvantages of each. An analysis of the asymptotic complexity of each is required as part of this assignment.

The knapsack problem is a combinatorial optimization problem. Given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity, the goal is to determine the number of each item to include in the knapsack such that the total weight does not exceed the capacity and the total value is maximized (Shaffer, 2011).

Brute Force Approach

The brute force method evaluates every possible combination of items to determine the one that provides the maximum value without exceeding the weight capacity (GeeksforGeeks, 2024). This approach guarantees finding the optimal solution by exploring all subsets of the given items.

Advantages:

· The brute force method ensures the optimal solution is found since every possible combination is evaluated

· The algorithm is straightforward to implement and understand

Disadvantages:

· The brute force approach has an asymptotic complexity of O(2n), where n is the number of items. This makes it impractical for large n, as the time required grows exponentially

· The method is highly inefficient for large datasets due to the need to evaluate every possible subset (GeeksforGeeks, 2024).

Dynamic Programming Approach

The dynamic programming (DP) approach solves the knapsack problem by breaking it down into smaller subproblems and storing the results of these subproblems to avoid redundant calculations (GeeksforGeeks, 2024). The most common DP solution for the 0/1 knapsack problem uses a 2D table where the entry DP[i][w] represents the maximum value that can be obtained with the first i items and a knapsack capacity w.

Advantages:

· The DP approach has an asymptotic complexity of O(nW), which is much more efficient than the brute force approach for large n

· By storing intermediate results, the DP approach avoids redundant calculations and speeds up the solution process

Disadvantages:

· The DP approach requires a 2D table of size (n+1)×(W+1), which can consume significant memory, especially for large values of n and W

· The algorithm is more complex to implement compared to the brute force approach, requiring careful handling of subproblems (GeeksforGeeks, 2024).

Asymptotic Complexity

· **Brute Force:** O(2n) — Evaluates every possible subset of items, leading to exponential growth in computation time as n increases.

· **Dynamic Programming:** O(nW) — More efficient for larger datasets, with computation time proportional to the product of the number of items and the maximum weight capacity.

While the brute force method guarantees the optimal solution, its exponential time complexity makes it impractical for large instances. The dynamic programming approach, though more complex, provides a polynomial time solution, making it suitable for larger datasets despite its higher memory requirements.

References

GeeksforGeeks. (2024). Difference between Brute Force and Dynamic Programming. https://www.geeksforgeeks.org/difference-between-brute-force-and-dynamic-programming/

Shaffer, C., A. (2011). A Practical Introduction to Data Structures and Algorithm Analysis. http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

504 words

Permalink Show parent

Re: Week 5
by Michael Oyewole - Monday, 22 July 2024, 4:55 PM

Hi Akomolafe,
Thank you for your post. One well-known optimization issue in computer science is the Kapsack problem. It demands that a knapsack with a capacity of W be able to supply the best feasible total value given the weights and values of n items.
Thanks
46 words

Permalink Show parent

Re: Week 5
by Romana Riyaz (Instructor) - Monday, 22 July 2024, 9:01 PM

Akomolafe,
Thank you for your response. Your description of the knapsack problem and the comparison between the brute force and dynamic programming approaches is clear and well-structured. Your explanation of the knapsack problem is concise and accurate. You might consider adding a brief example to illustrate the problem, which can help in visualizing the concept. Additionally, you could mention that brute force can be a helpful way to understand the problem's structure before optimizing it. You might also mention that the dynamic programming approach provides a systematic way to handle overlapping subproblems, which is a key aspect of its efficiency. Including a simple example (e.g., a knapsack with a capacity of 10 and items with different weights and values) can illustrate how each approach works. However, the reference you have added are not in correct APA format.
Best,
Romana
139 words

Permalink Show parent

Re: Week 5
by Moustafa Hazeen - Tuesday, 23 July 2024, 4:35 AM

Akomolafe, your submission provides a clear and concise explanation of the knapsack problem, comparing brute force and dynamic programming approaches effectively. You've supported your points with references and provided a good analysis of asymptotic complexities for both methods. To enhance it further, consider including a brief example or illustration to illustrate how each algorithm works in practice. Overall, well done!
60 words

Permalink Show parent

Re: Week 5
by Michael Oyewole - Monday, 22 July 2024, 4:29 PM

Knapsack Problem
An illustration of a combinational optimization problem is the Knapsack problem. Another name for this problem is the "Rucksack Problem." The problem's name derives from the maximizing problem in the scenario where a set of things with respective weights and values are placed in a bag with a maximum weight capacity of W (Introduction to Knapsack Problem, 2024). Choose how many of each item to include in a collection so that the total worth is maximized and the overall weight is less than the capacity.

Brute force
1.It provides a solution to an issue by applying the simplest technique possible. It usually doesn't solve the problem perfectly or be adaptable to new developments, but it does the job.
2. Trying every optimal solution to arrive at the final result is the quintessential example of brute force programming (Difference between Brute Force and Dynamic Programming, n.d.).
3. Brute force programming examines every conceivable route combination, however when a huge number of test cases is involved, different mathematical approaches produce faster answers.
4. Brute force approaches are often rarely employed in the industrial field since they slow down the software or product overall and are not the best use of time or space.

Advantages
1. Simplicity: It is easy to comprehend and apply the brute force method.
2. Completeness: Because it considers every potential combination of things, it ensures that the best solution will be found.

Disadvantages
1. Inefficiency: Because the brute force method must analyze every potential combination of elements, it is exceedingly inefficient for huge datasets.
2. Scalability: Because of its exponential development in the number of combinations, it does not scale well as the number of things increases.

Asymptotic complexity of the brute force approach
The asymptotic complexity of the brute force approach is O(2^n), when n is the number of elements. This is due to the fact that it generates 2^n combinations by treating every item as either being in the knapsack or not.

Dynamic Programming
1.The dynamic programming method divides the problem into progressively smaller feasible subproblems, much like divide and conquer. Dynamic programming is, to put it simply, an optimization over plain recursion (Difference between Brute Force and Dynamic Programming, n.d.).
2. These subproblems, however, are not resolved separately, in contrast to divide and conquer.
3. Rather, the solutions to these more manageable subproblems are retained and applied to determine overlap or similarity.

Advantages
1. Efficiency: Because dynamic programming avoids recalculating the same subproblems, it is far more efficient than brute force in solving the knapsack problem.
2. Optimality: By constructing the answer from previously computed variables, it also ensures an optimal solution.

Disadvantages
1. Memory Usage: In order to record the values of subproblems, dynamic programming needs more memory, which might build up, especially for larger problems.
2. Complexity: Compared to a brute force method, the implementation of a dynamic programming solution is more difficult.

Asymptotic complexity of the dynamic programming
The asymptotic complexity of the dynamic programming algorithm for the knapsack problem is O(nW), where W is the knapsack's capacity and n is the number of items. This is because it calculates the answer for each of the n elements at every weight up to W.

Brute Force Example

Example of brute force for finding Fibonacci of 15 using C++.

```
#include using namespace std;

int Fibonacci(int n){
if(n==1)
return 1;
if(n==2)
return 2;
return Fibonacci(n-1)+Fibonacci(n-2);
}
int main() {

int z=15;
cout< }
```

Output
987
(Geekforgeeks, n.d.)

Dynamic Programming Example
Example of dynamic programming for finding Fibonacci of 15 using C++.

```
#include using namespace std;

int Fibonacci(int n,vector&dp){
if(n==1)
return 1;
if(n==2)
return 2;
if(dp[n]!=-1)
return dp[n];
return dp[n]= Fibonacci(n-1,dp)+Fibonacci(n-2,dp);
}
int main() {
int z=15;
vectordp(z+1,-1);
cout< }
```

Output
987
(Geekforgeeks, n.d.)

References
Difference between Brute Force and Dynamic Programming. (n.d.). Geeksforgeeks. https://www.geeksforgeeks.org/difference-

between-brute-force-and-dynamic-programming/

Introduction to Knapsack Problem, its Types and How to solve them. (2024 July 06). Geekforgeeks. https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

648 words

Permalink Show parent

Re: Week 5

by Fadi Al Rifai - Wednesday, 24 July 2024, 2:50 PM

Hi Michael,
Good work, Thanks for sharing your explanation about the 'knapsack problem', and I like your description of using brute force and a dynamic programming algorithm.
Keep it up.

30 words

Permalink Show parent

Re: Week 5

by Nour Jamaluddin - Tuesday, 23 July 2024, 3:34 AM

The knapsack problem is a classic optimization problem in computer science and mathematics. As an example, imagine there is a thief (or a meticulous adventurer) with a knapsack of limited capacity (weight or volume). He has stumbled upon a treasure trove of loot, each item having a specific value and weight. He aims to fill the knapsack with the most valuable loot possible without exceeding the weight limit.

There are several variations of the knapsack problem, including the 0/1 knapsack problem (where each item can either be included or excluded) and the fractional knapsack problem (where items can be broken into smaller parts).

There are two main approaches to solving the knapsack problem, the Brute Force Approach and the Dynamic Programming Approach.

- Brute Force Approach

A brute force approach to the knapsack problem would be like trying every single way to fill the knapsack. You'd consider including each item or not, essentially making a yes/no decision for every item. This can be implemented with recursion, but for a large number of items, it becomes the most complicated way.

Advantages:

- Simplicity:

The brute force approach is straightforward to implement and understand.

- Guaranteed Solution:

It explores all possible combinations, so it is guaranteed to find the optimal solution if there is one.

Disadvantages:

- Exponential Time Complexity:

The number of combinations to explore grows exponentially with the number of items

- Repetitive Calculations:

The brute force approach might end up calculating the value of subproblems (combinations of a subset of items) multiple times, leading to wasted effort and time.

Asymptotic Complexity

The time complexity of the brute force approach is O(2^n), where n is the number of items. This is because there are 2^n possible subsets of n items.

- Dynamic Programming Approach

The dynamic programming approach solves the knapsack problem by breaking it down into smaller subproblems and storing the results of these subproblems to avoid redundant calculations (GeekforGeeks, 2024).

Advantages:

- Efficiency:

The dynamic programming approach significantly reduces the time complexity compared to brute force. It avoids redundant calculations by storing intermediate results.

- Optimal Solution:

Like the brute force method, dynamic programming guarantees an optimal solution.

Disadvantages:

- Increased Space Complexity:

Dynamic programming requires additional space to store the subproblem solutions in a table.

- Slightly More Complex Logic:

The dynamic programming approach involves setting up the table and understanding how to use the stored solutions, which can be slightly more complex to grasp than the brute force method.

Asymptotic Complexity

The time complexity of the dynamic programming approach is O(nW), where n is the number of items and W is the maximum weight capacity of the knapsack. The space complexity is also O(nW) due to the need to store the intermediate results in a table.

In short, we can decide the appropriate approach based on the purpose we want as follows:

- Implementation: The brute force approach is easier to implement and understand but is not feasible for large datasets. Dynamic programming requires more sophisticated implementation but is practical for larger instances.

- Space Requirements: While dynamic programming reduces time complexity, it increases space complexity due to the need for storing a table of intermediate results. The brute force approach, although time-intensive, does not have the same space requirements.

References

GeeksforGeeks. 2024. *Dynamic Programming or DP*. https://www.geeksforgeeks.org/dynamic-programming/

544 words

Permalink Show parent

Re: Week 5
by Fadi Al Rifai - Wednesday, 24 July 2024, 2:51 PM

Hi Nour,
Your post was well-detailed about the 'knapsack problem', and I like your description of using brute force and a dynamic programming algorithm.
Keep it up.
27 words

Permalink Show parent

Re: Week 5
by Jerome Bennett - Thursday, 25 July 2024, 9:29 AM

You did well in explaining the Knapsack problem. You also gave reasonable explanations on how Brute force and Dynamic programming could tackle this problem. Going forward, please use the correct format for your APA reference.
35 words

Permalink Show parent

Re: Week 5
by Winston Anderson - Tuesday, 23 July 2024, 4:59 AM

Knapsack Problem

The knapsack problem is a classic example of a combinatorial optimization problem. The problem can be described as follows: Given a set of items, each with a specific weight and value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit, and the total value is maximized. The most common variant is the 0/1 knapsack problem, where each item can either be included in the knapsack or not (hence the name 0/1) (GeeksforGeeks, 2024; Wikipedia, 2024).

Brute Force vs. Dynamic Programming

Brute Force Approach

The brute force approach involves evaluating all possible combinations of items to find the one that maximizes the total value without exceeding the weight limit. This method guarantees finding the optimal solution since it explores every possible subset of items.

Advantages:

- Simplicity: The brute force method is straightforward and easy to understand and implement.

- Guaranteed Optimal Solution: Since it evaluates all possible combinations, it ensures that the optimal solution is found if it exists.

Disadvantages:

- Inefficiency: The time complexity of the brute force approach is exponential, specifically ; O(2^n); , where ; n ; is the number of items. This makes it impractical for large datasets.

- High Computational Resource Usage: It requires significant computational resources, including time and memory, making it unsuitable for real-world applications with large numbers of items (GeeksforGeeks, 2024; Linkedin, n.d.).

Dynamic Programming Approach

Dynamic programming (DP) offers a more efficient solution by breaking down the problem into smaller subproblems and storing their solutions to avoid redundant calculations. The 0/1 knapsack problem can be solved using a DP approach with a time complexity of ; O(nW); , where ; n ; is the number of items and ; W ; is the maximum weight capacity of the knapsack.

Advantages:

- Efficiency: The DP approach significantly reduces the time complexity compared to brute force, making it feasible for larger datasets.

- Optimal Solution: Like the brute force method, DP also guarantees finding the optimal solution.

- Reuse of Subproblem Solutions: By storing solutions to subproblems, DP avoids redundant calculations, improving efficiency (GeeksforGeeks, 2024; Winnie, 2023; Khandelwal, 2021).

Disadvantages:

- Space Complexity: The DP approach requires additional space to store the solutions of subproblems, leading to a space complexity of ; O(nW); .

- Complexity in Implementation: The DP approach is more complex to implement compared to brute force, requiring a good understanding of the problem and the DP technique (GeeksforGeeks, 2024; Khandelwal, 2021).

Example

Consider a knapsack with a weight limit of 50 and three items with the following weights and values:

- Item 1: Weight = 10, Value = 60

- Item 2: Weight = 20, Value = 100

- Item 3: Weight = 30, Value = 120

Brute Force Solution

For the brute force solution, we would evaluate all possible combinations of items:

1. No items: Total weight = 0, Total value = 0

2. Item 1: Total weight = 10, Total value = 60

3. Item 2: Total weight = 20, Total value = 100

4. Item 3: Total weight = 30, Total value = 120

5. Items 1 and 2: Total weight = 30, Total value = 160

6. Items 1 and 3: Total weight = 40, Total value = 180

7. Items 2 and 3: Total weight = 50, Total value = 220

8. Items 1, 2, and 3: Total weight = 60, Total value = 280 (exceeds weight limit)

The optimal solution is to include items 2 and 3, giving a total value of 220 without exceeding the weight limit.

Dynamic Programming Solution

Using dynamic programming, we create a table to store the maximum value obtainable for each subproblem. The table is filled in a bottom-up manner.

DP Table Initialization:

Items/Weight	0	10	20	30	40	50
0	0	0	0	0	0	0
1	0	60	60	60	60	60
2	0	60	100	160	160	160
3	0	60	100	160	180	220

The maximum value obtainable with a weight limit of 50 is 220, achieved by including items 2 and 3.

Conclusion

In summary, while the brute force approach is simple and guarantees finding the optimal solution, it is computationally expensive and impractical for large datasets. On the other hand, the dynamic programming approach is more efficient and suitable for larger problems, though it requires additional space and a more complex implementation. The choice of method depends on the specific requirements and constraints of the problem at hand.

References

GeeksforGeeks. (2024, March 26). *Difference between Brute Force and Dynamic Programming*. GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-brute-force-and-dynamic-programming/

GeeksforGeeks. (2024, May 30). *01 Knapsack Problem*. GeeksforGeeks. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

GeeksforGeeks. (2024, July 6). *Introduction to Knapsack Problem, its Types and How to solve them*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

Khandelwal, V. (2021, November 9). *Knapsack Problem: 0-1 and Fractional Using Dynamic Programming*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/knapsack-problem

Linkedin. (n.d.). *What are the advantages and disadvantages of using a brute force algorithm?* https://www.linkedin.com/advice/1/what-advantages-disadvantages-using-brute-oruof

w3schools. (n.d.). *DSA The 0/1 Knapsack Problem*. W3schools. https://www.w3schools.com/dsa/dsa_ref_knapsack.php

Wikipedia. (2024, May 13). *Knapsack problem*. https://en.wikipedia.org/wiki/Knapsack_problem

Winnie, O. (2023b, December 17). *Knapsack Problem: Brute force vs Dynamic Programming?* https://www.linkedin.com/pulse/knapsack-problem-brute-force-vs-dynamic-programming-ouma-winnie-8fvtf

853 words

Permalink Show parent

Re: Week 5

by Anthony Jones - Tuesday, 23 July 2024, 8:40 AM

The (0/1) knapsack problem is a problem of optimization. Essentially, we have several items with varying values and cost. and we can only include items up to a certain cost, so we need to find a way to maximize the value. This is the heart of the knapsack problem. It is often presented as trying to fill a knapsack that can only hold so much weight. If we put an item in that weighs more than another item, we won't be able to fit as much, however, it may be more valuable to include it. For instance, consider a knapsack that can hold cost 5 and have the following items:

	Value	Cost
Item1	2	1
Item2	3	3
Item3	5	4
Item4	3	2
Item5	6	5

It can seem most intuitive to rank the items by least cost and fill the knapsack as much as we can: [Item1, Item4] giving us a total value of 5. But we can do better. We could try ranking the items by value and adding them: [Item 5] giving us a total value of 6. But this still isn't optimal, so how do we do it?

Brute Force

We can try a brute force method where we try every combination. Here's some pseudocode for this:

```
function Knapsack(items, capacity){
    initialize combination array
    for every item in items{
        combination = Knapsack(items-item, capacity-item.weight)
        combination += item
        combination.value += item.value
        combinations.append(combination)
    }
    return combination with maximum value
}
```

This will get us the optimum combination, but it's really expensive. It runs in time. For small problems this is fine, but becomes really inefficient for larger problem sizes, even with simply 15 items! Of course this could definitely be simplified so that it evaluates combinations instead of permutations (recursion method), but it is still inefficient (GeeksforGeeks, 2012).

Greedy approach

Sort the algorithm by the value-cost ratio and include as much as possible: [Item1, Item4] giving us a total value of 5. However, this isn't the optimum as we have discussed. However, the algorithm is very fast and works well in larger problem sizes. The reason why it doesn't always provide the optimum result is because it is specifically designed for the fractional knapsack where we can take only part of an item (Wenk, 2018). If we can do that, the optimum value becomes 7.5. However, since we can't split item3, we are left with unused space of 2 in the knapsack.
Greedy algorithms, however, are an indicator of a Dynamic programming approach.

Dynamic programming

The dynamic programming approach is quite interesting. Assuming we have n items and a capacity of W, we create an n × W table for storing best value: (items sorted by increasing cost)

W=	**1**	**2**	**3**	**4**	**5**
Item1	0	0	0	0	0
Item4	0	0	0	0	0
Item2	0	0	0	0	0
Item3	0	0	0	0	0
Item5	0	0	0	0	0

We iteratively fill the table by evaluating if it is best to include the item for a given weight based on the previous best value for the weight. If we are to add the item, we put the new total value in the cell. The first row is pretty straightforward: it's best to include the item as soon as we can add it because there are no better alternatives:

W=	**1**	**2**	**3**	**4**	**5**
Item1	2	2	2	2	2

The next one is pretty similar. We cannot add it for the first capacity, but once we get to 2, we can add it, and since it has more value than the previous item, it gets added, instead of the other item

W=	1	2	3	4	5
Item1	2	2	2	2	2
Item4	2	3	5	5	5

It is important to realize here that the rows in the table represent the best value of the items that come before the item as well as the item itself. Item 2 doesn't improve the value at all because the algorithm decides that it is best not to include it since it never provides a higher value than the value in the previous row for the capacity - the Item2's cost (example: row 4 if we include item2, we would have a value of 3 + 2 (the value in the previous row for the capacity remaining after we add item2: 1). Since this is not more than the existing best for capacity 4, we do not add it)

W=	1	2	3	4	5
Item1	2	2	2	2	2
Item4	2	3	5	5	5
Item2	2	3	5	5	5

Item3 is interesting because when we get to capacity 5, it allows us to add it while still adding item 2, so we can add it and increase the best cost for capacity 5 to 7:

W=	1	2	3	4	5
Item1	2	2	2	2	2
Item4	2	3	5	5	5
Item2	2	3	5	5	5
Item3	2	3	5	5	7

Item 5 is never added, because it can never improve the best value:

W=	1	2	3	4	5
Item1	2	2	2	2	2
Item4	2	3	5	5	5
Item2	2	3	5	5	5
Item3	2	3	5	5	7
Item4	2	3	5	5	7

The important thing to understand is how the algorithm determines whether to include the item. It considers the expression: *highestValue[Row, Capacity] = max(Item.value + highestValue[Row-1, Capacity-Item.cost], highestValue[Row-1, Capacity]*. Essentially, it decides if its best to include it or leave it out based on the whether it woould improve the value

For our algorithm we can see that we got the final best value of 7. This is the best solution for the (0/1) Knapsack problem. The dynamic programming approach has time complexity of where W represents the capacity and n represents the number of items. One thing that should be noted, however, is that actual code implementations of the DP approach include a row and column for no items included and no capacity.

Overall, the dynamic programming solution provides a robust solution to a difficult problem. Ultimately, it is a memoised algorithm that recursively evaluates if there is a higher value for not including the item (i) or for including i with the best value for the remaining space. Dynamic programming is often faster than other methods but doesn't always apply to situations and is often conceptually difficult to implement.

References

freeCodeCamp.org. (2020, December 3). *Dynamic programming*. YouTube. https://youtu.be/oBt53YbR9Kk

GeeksforGeeks. (2012, March 19). *0/1 knapsack problem*. GeeksforGeeks. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/#

Reducible. (2020, August 16). *5 simple steps for solving dynamic programming problems*. YouTube. https://youtu.be/aPQY__2H3tE

Wenk, C. (2018). Greedy algorithms: Knapsack problem. In *CMPS 6610 Algorithms -Fall*. https://www.cs.tulane.edu/~carola/teaching/cmps6610/fall18/slides/Knapsack.pdf

WilliamFiset. (2017, September 16). *0/1 knapsack problem | dynamic programming*. YouTube. https://youtu.be/cJ21moQpofY

1145 words

Permalink Show parent

Re: Week 5
by Benjamin Chang - Tuesday, 23 July 2024, 7:46 PM

Hi Anthony

In this week's post, you have demonstrated a comprehensive comprehension of the differences between brute force and dynamic programming by utilizing time complexity to analyze these two algorithms. Additionally, you provide a detailed explanation of the process of iteratively evaluating each item in the matrix. Well done!

Yours sincerely
Benjamin
52 words

Permalink Show parent

Re: Week 5
by Naqaa Alawadhi - Tuesday, 23 July 2024, 2:22 PM

The knapsack problem involves selecting a combination of items with specific weights and values to maximize the value within a given weight constraint. Brute force involves trying all possible combinations to find the optimal solution, while dynamic programming breaks down the problem into subproblems and stores solutions to avoid redundant calculations.

Comparison:

- Brute Force:
- Advantages: Guarantees finding the optimal solution. Simple to implement for small inputs.
- Disadvantages: Inefficient for large inputs due to its exponential time complexity.
- Asymptotic Complexity: O(2^n), where n is the number of items.

- Dynamic Programming:
- Advantages: Efficient for larger inputs by avoiding redundant calculations. Provides optimal solutions.
- Disadvantages: Requires more memory due to storing solutions for subproblems.
- Asymptotic Complexity: O(nW), where n is the number of items and W is the maximum weight.

Example:

Consider a knapsack with a weight capacity of 10 and three items:
1. Item A: Weight = 4, Value = $10
2. Item B: Weight = 6, Value = $15
3. Item C: Weight = 3, Value = $7

- Brute Force Approach: Try all possible combinations (2^3 = 8) to find the optimal solution.
- Dynamic Programming Approach: Create a table to store intermediate results for subproblems and build up towards the optimal solution efficiently.

References:
Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

Smith, M. D., & Smith, J. M. (2011). Dynamic Programming and Optimal Control. Athena Scientific.
246 words

Re: Week 5

by Siraajuddeen Adeitan Abdulfattah - Tuesday, 23 July 2024, 8:29 PM

The knapsack problem also known as the Rucksack problem is a combinational optimization problem (Geeksforgeeks, 2024) where a set of items with known weights or values is to be packed into a container with a maximum capacity, such that the total items that would fit into the container must be a subset of items of maximum total weight/value that is less than the container's maximum capacity. (Google OR-Tools, 2023).

The knapsack problem is classified into four types as listed below:

Fractional knapsack problem

0/1 Knapsack problem

Bounded knapsack problem

Unbounded knapsack problem (Geeksforgeeks, 2024)

Let's consider an example of the 0/1 knapsack problem;

A treasure hunter with a backpack (Knapsack) of a weight limit 10Kg, is in a treasure room containing the following treasures; Globe (weight; 1Kg, value; $200), Gold crown (weight; 3Kg, value; $500), Microscope (weight; 2Kg, value; $300) and Trophy (weight; 5Kg, value; $400) (DSA The 0/1 Knapsack Problem, n.d).

Some of the applications of the solution to knapsack 0/1 problem include profit maximization for businesses while reducing cost (without overspending), optimizing loading of goods to ensure high priority goods, and/or most valuable goods are loaded in logistics company, help in deciding what projects to fund without exceeding a budget. (DSA The 0/1 Knapsack Problem, n.d).

The rules of engagement are follows:

Each item has a value and weight.

Knapsack has a weight limit.

You have to choose items that will fit into the knapsack considering the weight limit.

You have an option to take an item or not, you can't take half of an item.

Goal:

The goal is to maximize the total value of the items in the knapsack (DSA The 0/1 Knapsack Problem, n.d).

Here's a brute force approach to solving the problem:

Brute force checks all the possibilities while looking for the best result and it requires the most calculations but it is straight forward.

Here's how to solve the 0/1 knapsack problem with brute force;

Calculate the value of every possible combination of items/treasure that can fit into the knapsack

Discard the combinations that are bigger/heavier than the knapsack weight limit

Choose the combination of treasures/items with the highest total value (DSA The 0/1 Knapsack Problem, n.d).

Algorithm description:

Consider individual item one at a time

If there is capacity left for the current treasure, add it by adding the value and reducing the remaining capacity with its weight. Then call the function on itself for the next treasure/item

Don't add the current treasure before calling the function on itself for the next treasure.

Return the maximum value from the two scenarios above (adding the current treasure or not adding it) (DSA The 0/1 Knapsack Problem, n.d).

Code implementation (Brute force):

Solving the 0/1 knapsack problem with recursion and brute force;

```
def knapsack_brute_force(capacity, n):
    print(f"knapsack_brute_force({capacity},{n})")
    if n == 0 or capacity == 0:
        return 0

    elif weights[n-1] > capacity:
        return knapsack_brute_force(capacity, n-1)

    else:
        include_item = values[n-1] + knapsack_brute_force(capacity-weights[n-1], n-1)
        exclude_item = knapsack_brute_force(capacity, n-1)
        return max(include_item, exclude_item)

values = [300, 200, 400, 500]
weights = [2, 1, 5, 3]
capacity = 10
n = len(values)

print("\nMaximum value in Knapsack =", knapsack_brute_force(capacity, n))
```

(DSA The 0/1 Knapsack Problem, n.d).

Running the code example creates a recursion tree that looks like this, each gray box represents a function call:

Running the code means that the knapsack_brute_force function is called many times recursively. You can see that from all the printouts.

Each time the function is called, it will either include the current item n-1 or not.

The print statement shows us each time the function is called.

If we run out of items to check (n==0), or we run out of capacity (capacity==0), we do not do any more recursive calls because no more items can be added to the knapsack at this point.

If the current item is heavier than the capacity (weights[n-1] > capacity), forget the current item and go to the next item.

If the current item can be added to the knapsack, see what gives you the highest value: adding the current item, or not adding the current item (DSA The 0/1 Knapsack Problem, n.d).

Dynamic programming Algorithm /The Tabulation Approach (bottom-up):

Here is an iterative approach to solving the 0/1 knapsack problem, it is technique used in dynamic programming;

It follows the bottom to top approach by first filling up a table with the result from the smallest or most basic subproblems and then fill the next table values using the previous result.

Descriptions:

Take one treasure at a time and increase the knapsack capacity from 0 to the knapsack limit.

If the current treasure is not too heavy, check what gives the highest value: add it or not. Then store the maximum of the two values in the table.

If the current treasure is too heavy to be added, use the previously calculated value at the current capacity where the current treasure was not considered (DSA The 0/1 Knapsack Problem, n.d).

Code implementation:

Solution to 0/1 knapsack problem using tabulation:

```
def knapsack_tabulation():
```

```
    n = len(values)
```

```
    tab = [[0]*(capacity + 1) for y in range(n + 1)]

    for i in range(1, n+1):
        for w in range(1, capacity+1):
            if weights[i-1] <= w:
                include_item = values[i-1] + tab[i-1][w-weights[i-1]]
                exclude_item = tab[i-1][w]
                tab[i][w] = max(include_item, exclude_item)
            else:
                tab[i][w] = tab[i-1][w]

    for row in tab:
         print(row)
    return tab[n][capacity]

values = [300, 200, 400, 500]
weights = [2, 1, 5, 3]
capacity = 10
print("\nMaximum value in Knapsack =", knapsack_tabulation())
```

(DSA The 0/1 Knapsack Problem, n.d).

If the item weight is lower than the capacity it means it can be added. Check if adding it gives a higher total value than the result calculated in the previous row, which represents not adding the item. Use the highest (max) of these two values. In other words: Choose to take, or not to take, the current item.

This line might be the hardest to understand. To find the value that corresponds to adding the current item, we must use the current item's value from the values array. But in addition, we must reduce the capacity with the current item's weight, to see if the remaining capacity can give us any additional value. This is similar to check if other items can be added in addition to the current item, and adding the value of those items.

In case the current item is heavier than the capacity (too heavy), just fill in the value from the previous line, which represents not adding the current item (DSA The 0/1 Knapsack Problem, n.d).

Time complexity

Brute force approach; being the slowest has a time complexity of O(2^n), n is the number of possible items that can be packed. It means the number of computations doubles for each extra item that needs to be considered.

Tabulation approach; has a time complexity of O(n . C), n is the number of items and C is the knapsack capacity. It is the most favorable approach because it offers a predictable in memory usage and how it runs (DSA The 0/1 Knapsack Problem, n.d).

Reference(s):

Geeksforgeeks, (July, 2024). Introduction to Knapsack Problem, its Types and How to solve them. Retrieved from: https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

Google OR-Tools, (2023). The Knapsack Problem. Retrieved from:https://developers.google.com/optimization/pack/knapsack

DSA The 0/1 Knapsack Problem, (n.d). DSA The 0/1 Knapsack Problem. Retrieved from: https://www.w3schools.com/dsa/dsa_ref_knapsack.php

1242 words

Permalink Show parent

Re: Week 5
by Wingsoflord Ngilazi - Thursday, 25 July 2024, 1:52 AM

A well-written post! I sincerely appreciate the effort you put into your work. Thank you for sharing some insightful examples.
20 words

Permalink Show parent

Re: Week 5
by Manahil Farrukh Siddiqui - Tuesday, 23 July 2024, 11:07 PM

The knapsack problem is a fundamental concept in combinatorial optimization. Imagine being a thief tasked with filling a knapsack of limited capacity (W) with items of various weights (w_i) and values (v_i). The objective is to maximize the items' total value (V) in the knapsack without exceeding its weight limit.

To tackle this problem, several approaches can be considered, including brute force and dynamic programming.

Brute Force: The brute force approach evaluates every possible combination of items. Each combination's total weight and value are calculated to determine the optimal selection that fits within the weight limit (FreeCodeCamp, 2020).

Advantages: This method is straightforward to understand.

Disadvantages: However, its simplicity comes with a high computational cost. As the number of items (N) increases, the number of possible combinations grows exponentially, with a time complexity of O(2^N). This makes brute force impractical for large datasets (FreeCodeCamp, 2020). Additionally, brute force often involves redundant calculations, as it repeatedly solves subproblems.

Example: Consider three items: a gold necklace (weight 2, value 10), a diamond ring (weight 1, value 8), and a ruby pendant (weight 3, value 7). If the knapsack capacity is four units, brute force will explore all possible combinations, totalling eight scenarios. Ultimately, the optimal solution is the combination of the necklace and the ring, giving a total value of 18.

Dynamic Programming: Dynamic programming offers a more systematic and efficient approach by breaking the problem into smaller, overlapping subproblems (Patil, 2023).

Advantages: This method avoids repeated calculations by solving each subproblem once, significantly improving time efficiency to O(NW) (Vaia, n.d.). It also uses a table to store intermediate results, leading to a space complexity of O(NW) (Vaia, n.d.).

Disadvantages: The trade-off is the space required for the table, which can be significant for large datasets. Implementing dynamic programming requires managing complex table structures (Patil, 2023).

Example: Using dynamic programming, we create a table where each cell (i, w) represents the maximum value achievable with the first i items and a weight limit of w. By systematically filling this table, we find that the optimal solution, like the brute force approach, is the necklace and ring combination, yielding a value of 18. Dynamic programming simplifies the process as each subproblem is solved only once.

In conclusion, the knapsack problem illustrates the balance between simplicity and effectiveness. While brute force is easy to understand, its exponential time complexity makes it unsuitable for large datasets. With its systematic approach, dynamic programming offers a more efficient solution, making it ideal for real-world scenarios with many items. This method provides a structured and rapid way to solve complex problems.

References

FreeCodeCamp. (2020, January 6). Brute Force Algorithms Explained. FreeCodeCamp.org. https://www.freecodecamp.org/news/brute-force-algorithms-explained/

Patil, R. (2023, August 3). Efficient Problem-solving with Dynamic Programming. Medium. https://medium.com/@rahulptl556/efficient-problem-solving-with-dynamic-programming-a87b9301c13e

Vaia. (n.d.). Knapsack Problem: 0/1 & Unbounded Variants. Vaia. Retrieved May 13, 2024, from https://www.vaia.com/en-us/explanations/computer-science/algorithms-in-computer-science/knapsack-problem/

471 words

Permalink Show parent

Re: Week 5

by Muritala Akinyemi Adewale - Tuesday, 23 July 2024, 11:17 PM

The Knapsack Problem: A Balancing Act

The knapsack problem is a classic optimization problem in computer science. Imagine you're a thief planning a heist and have a backpack (knapsack) with a limited capacity (weight limit). You want to steal the most valuable items (maximum value) from a vault, but each item has a specific weight. The challenge lies in selecting the optimal combination of items that maximizes your total value without exceeding the knapsack's weight limit.

Brute Force Approach: Simple but Inefficient

A brute-force approach to the knapsack problem involves trying out every single possible combination of items. Here's how it works:

1. **Generate all combinations:** List all possible subsets of items that can be picked from the available items.
2. **Calculate total value and weight:** For each combination, calculate the total value of the items and their combined weight.
3. **Select the best combination:** Choose the combination with the highest total value that doesn't exceed the knapsack's weight limit.

Example:

Let's say you have three items:

- Item 1: Value = $10, Weight = 5 kg
- Item 2: Value = $15, Weight = 3 kg
- Item 3: Value = $20, Weight = 7 kg

The knapsack's weight limit is 10 kg.

A brute-force approach would involve listing all eight possible combinations (including the empty set) and checking their weight and value. The optimal solution would be Item 1 and Item 2, with a total value of $25 and a weight of 8 kg, within the limit.

Advantages:

- Easy to understand and implement.

Disadvantages:

- **Computationally expensive:** The number of combinations grows exponentially with the number of items. This becomes impractical for even moderate-sized problems.
- **Asymptotic Complexity:** O(2^n), where n is the number of items. This means the execution time doubles with each additional item, making it inefficient for large datasets.

Dynamic Programming: Efficient for Overlapping Subproblems

Dynamic programming offers a more efficient solution for the knapsack problem. It leverages the concept of overlapping subproblems: smaller subproblems within the larger problem that are solved repeatedly. Dynamic programming solves these subproblems once and stores the results in a table to avoid redundant calculations.

Steps:

1. **Create a table:** Initialize a table where each cell represents the maximum value achievable for a given weight capacity and a subset of items (up to a certain point).
2. **Fill the table:** Iteratively fill the table, considering each item and its potential contribution to the value within the weight limit. Utilize previously calculated values from the table for subproblems.
3. **Trace back to find the solution:** Starting from the desired weight limit in the table, trace back to identify the items included in the optimal solution.

Example:

Using the same items and weight limit from the brute-force example, dynamic programming would build a table, calculate the maximum value at each weight capacity for possible item combinations, and then trace back to find the optimal solution (Item 1 and Item 2).

Advantages:

- **More efficient:** Solves overlapping subproblems only once, significantly reducing computation time.
- **Suitable for larger problems:** Handles larger datasets more efficiently compared to brute force.

Disadvantages:

- **Slightly more complex to understand and implement:** Requires more planning and upfront work to design the table and logic.
- **Space complexity:** Requires additional memory to store the table, which can be a concern for very large datasets.

Asymptotic Complexity: O(n * W), where n is the number of items and W is the weight limit. This complexity is significantly better than the brute force approach for larger n.

Reference:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.)

590 words

Permalink Show parent

Re: Week 5
by Anthony Jones - Wednesday, 24 July 2024, 9:45 PM

Hello,

Good job!
Can you provide an example of the dynamic programming approach being used?
How can we optimize the algorithm?

God bless!

Anthony
24 words

Permalink Show parent

Re: Week 5

by Cherkaoui Yassine - Wednesday, 24 July 2024, 2:49 AM

The knapsack problem: is a classic optimization problem where a thief must determine the most valuable combination of items to steal without exceeding the weight capacity of their knapsack. The challenge is to maximize the total value of the items taken while staying within a given weight limit.

The brute force: method involves examining every possible subset of items to find the combination that yields the highest value without exceeding the weight limit. Here are the advantages and disadvantages:

Advantages:

- Completeness: The brute force approach guarantees finding the optimal solution since it evaluates all possible combinations.

Disadvantages:

- Time Complexity: The brute force method has an exponential time complexity of O(2^n), where n is the number of items. This makes it impractical for large datasets.

- Inefficiency: It performs redundant calculations and is highly inefficient in terms of computational resources.

Example: Consider a knapsack with a weight limit of 4 pounds and the following items:

- Item 1: Weight = 1, Value = 1

- Item 2: Weight = 2, Value = 2

- Item 3: Weight = 3, Value = 3

Using brute force, we evaluate all combinations:

1. No items (Value = 0)

2. Item 1 (Value = 1)

3. Item 2 (Value = 2)

4. Item 3 (Value = 3)

5. Item 1 and Item 2 (Value = 3)

6. Item 1 and Item 3 (exceeds weight limit)

7. Item 2 and Item 3 (exceeds weight limit)

8. Item 1, Item 2, and Item 3 (exceeds weight limit)

The optimal solution is taking Item 2 and Item 1 with a total value of 3.

Dynamic programming: solves the knapsack problem by breaking it down into smaller subproblems and solving each subproblem just once, storing the solutions in a table to avoid redundant calculations.

Advantages:

- Efficiency: It significantly reduces the number of computations by storing results of subproblems, leading to a polynomial time complexity of O(nW), where n is the number of items and W is the weight capacity of the knapsack.

- Optimality: It guarantees an optimal solution while being more efficient than brute force.

Disadvantages:

- Memory Usage: Requires additional memory to store the table of subproblem solutions.

- Complexity: The implementation can be more complex compared to the brute force approach.

Example:

Using the same items and knapsack capacity:

- Initialize a table with dimensions (n+1) x (W+1), where n is the number of items and W is the weight capacity.

- Fill the table using the following logic:

- If an item is not included, the value is the same as the previous row for the same column.

- If an item is included, the value is the maximum of not including the item or including it and adding its value to the value obtained from the remaining capacity.

The optimal value is found at the bottom-right corner of the table, indicating the maximum value achievable with the given items and weight capacity.

Asymptotic Complexity

- Brute Force: O(2^n)

- Dynamic Programming: O(nW)

References:
Knapsack problem. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Knapsack_problem

509 words

Permalink Show parent

Re: Week 5
by Winston Anderson - Thursday, 25 July 2024, 4:29 AM

Hi Cherkaoui,
Your explanation of the knapsack problem and the comparison between the brute force and dynamic programming (DP) approaches is clear and well-structured. Overall, your explanation is informative and covers the essential points well.
35 words

Permalink Show parent

Re: Week 5
by Wingsoflord Ngilazi - Wednesday, 24 July 2024, 5:06 PM

In your own words describe the 'knapsack problem'. Further, compare and contrast the use of a brute force and a dynamic programming algorithm to solve this problem in terms of the advantage and disadvantages of each. An analysis of the asymptotic complexity of each is required as part of this assignment.

Include one or two examples to explain your thought process to show what is occurring and how the methodology works. Use APA citations and references for any sources used.

The Knapsack Problem

The knapsack problem is a basic problem in combinatorial optimization. The following is a description of the algorithm: Determine the quantity of each item to include in a collection given a set of objects that each have a weight and a value in order to keep the total weight within a predetermined range and the total value as high as feasible. The reason this problem is known as the "knapsack problem" is that it can be understood as an attempt to maximize the total value of objects while staying within the weight limit of a knapsack (GeeksforGeeks, 2024).

Brute Force Algorithm

The brute force approach to solving the knapsack problem involves generating all possible combinations of items and then checking each combination to see if it is valid (i.e., the total weight is within the limit) and then computing its total value. The combination with the highest value that fits within the weight limit is chosen (GeeksforGeeks, 2024).

Example

Imagine you are an adventurer setting out on a treasure hunt. You have a magical backpack that can carry a maximum weight of 20 units. You need to decide which treasures to take with you to maximize the total value of your collection. Here are the treasures available to you, each with its own weight and value:

Maximum Weight Capacity (W): 20 units

Treasures (items) with their respective weights and values:

Golden Crown: Weight 10, Value 60

Jeweled Sword: Weight 15, Value 100

Ruby Ring: Weight 5, Value 40

The brute force approach would check all combinations of these items:

1. Golden Crown (10 units, 60 gold coins)
2. Jeweled Sword (15 units, 100 gold coins)
3. Ruby Ring (5 units, 40 gold coins)

You check all 8 combinations of these treasures:

- No treasures: 0 value
- Golden Crown only: 60 value
- Jeweled Sword only: 100 value
- Ruby Ring only: 40 value
- Golden Crown and Jeweled Sword: Exceeds weight limit
- Golden Crown and Ruby Ring: 100 value
- Jeweled Sword and Ruby Ring: 140 value
- All three treasures: Exceeds weight limit

The optimal combination is the **Jeweled Sword and Ruby Ring** which provides the maximum value of 140 gold coins without exceeding the weight limit.

Advantages

Simple and easy to implement.

Guaranteed to find the optimal solution.

Disadvantages

Extremely inefficient for large inputs due to its exponential time complexity.

Complexity Analysis

The time complexity of the brute force approach is $O(2^n)$, where n is the number of items. This is because each item can either be included or excluded.

Dynamic Programming Algorithm

Dynamic Programming is a method in computer programming that efficiently addresses problems with overlapping subproblems and an optimal substructure property. When a problem can be broken down into subproblems, which in turn can be further divided, and these subproblems overlap, dynamic programming stores the solutions to these subproblems for future use. This technique improves CPU efficiency by preventing the repeated computation of the same subproblems, thereby finding the optimal solution more effectively (Programiz, n.d.).

Example

To solve the knapsack problem using dynamic programming, we create a table where the entry dp[i][w] represents the maximum value achievable with the first i items and a weight limit of w. We initialize the table with zeros, representing the case where no items are chosen or the weight limit is zero. We then iterate through each item and each possible weight limit, deciding whether to include each item based on its weight and value. If the item's weight is less than or equal to the current weight limit, we choose the maximum value between including the item and not including it (Programiz, n.d.). This systematic approach ensures that we consider all possible combinations efficiently, ultimately finding the maximum value that can be achieved without exceeding the weight limit. For the given problem, this method determines that the optimal solution is to take the Jeweled Sword and the Ruby Ring, maximizing the value at 140 gold coins.

Advantages

Efficient and avoids redundant calculations.

Suitable for large inputs where brute force is impractical.

Disadvantages

Requires additional memory for the table.

May not be as intuitive to implement as brute force.

Complexity Analysis

The time complexity of the dynamic programming approach is O(nW), where n is the number of items and W is the maximum weight capacity of the knapsack. The space complexity is also O(nW) due to the storage requirements for the table.

Conclusion

The brute force method is infeasible for big inputs due to its exponential time complexity, even though it ensures an optimal answer. However, the dynamic programming method uses more memory but provides a polynomial time solution, which makes it far more effective for larger problems.

References

Programiz. (n.d.). *Dynamic Programming*. Retrieved from https://www.programiz.com/dsa/dynamic-programming

GeeksforGeeks. (2024, July 6). *Introduction to Knapsack Problem, its Types and How to solve them*. Retrieved from https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

878 words

Permalink Show parent

Re: Week 5

by Anthony Jones - Wednesday, 24 July 2024, 10:05 PM

Hello,

Good post! Can you provide an example of the dp being used in order to better explain the concept.
How can we optimize it? Would sorting the items help? if so, how?

God bless!

Anthony

36 words

Permalink Show parent

Re: Week 5

by Winston Anderson - Thursday, 25 July 2024, 4:20 AM

Hi Wingsoflord,
Your explanation of the knapsack problem and the comparison between the brute force and dynamic programming (DP) approaches is generally clear and informative. Your explanation is solid and covers the essential points well.

35 words

Permalink Show parent

Re: Week 5

by Mejbaul Mubin - Wednesday, 24 July 2024, 6:10 PM

The knapsack problem is a classic optimization problem in which a set of items, each with a weight and a value, must be selected to include in a knapsack of limited capacity. The objective is to maximize the total value of the items in the knapsack without exceeding its weight capacity. This problem can be described mathematically as follows:

Given n items, each with a weight w_i and a value v^2 and a knapsack with a maximum weight capacity W, find a subset of the items such that the total weight does not exceed W and the total value is maximized.

Brute Force Approach

Description

The brute force approach involves evaluating all possible subsets of the items to determine which subset provides the maximum value without exceeding the weight capacity. This approach ensures that the optimal solution is found because it considers every possible combination.

Advantages

1. **Simplicity:** The brute force method is straightforward to implement and understand.

2. **Optimality:** It guarantees finding the optimal solution since it evaluates all possible combinations.

Disadvantages

1. Inefficiency: The brute force approach is computationally expensive. For n items, there are 2^n possible subsets, leading to an exponential time complexity.

2. Scalability: It becomes impractical for large numbers of items due to the exponential growth in the number of combinations.

Asymptotic Complexity

The time complexity of the brute force approach is O(2") because it evaluates all possible subsets. The space complexity is O(n) for the stack used in the recursive calls.

Example

Consider 3 items with weights and values as follows:

Item 1: weight 1, value 10

Item 2: weight 2, value 15

Item 3: weight 3, value 40

And a knapsack with a weight capacity of 5. The brute force approach would evaluate all 8 possible subsets (including the empty subset) and find that the subset {Item 2, Item 3} gives the maximum value of 55 without exceeding the weight capacity.

Dynamic Programming Approach

Description

The dynamic programming (DP) approach solves the knapsack problem by building a table to store the maximum value achievable with a given capacity. It iteratively builds solutions to subproblems and uses these solutions to construct the solution to the original problem.

Advantages

1. Efficiency: The dynamic programming approach is much more efficient than the brute force method. It significantly reduces the number of computations by storing and reusing intermediate results.

2. Scalability: It can handle larger instances of the problem compared to the brute force approach.

Disadvantages

1. Complexity: The DP approach is more complex to implement than the brute force method.

2. Space: It requires additional memory to store the DP table, which can be a limitation for very large input sizes.

Asymptotic Complexity

The time complexity of the dynamic programming approach is O(nW), where n is the number of items and W is the weight capacity of the knapsack. The space complexity is also O(nW) due to the storage of the DP table.

Example

Using the same items and knapsack capacity as the previous example:

• Create a table with rows representing items and columns representing weight capacities from 0 to 5.

• Fill the table by iterating through items and capacities, updating each cell with the maximum value achievable.

After filling the table, the maximum value for a knapsack of capacity 5 will be found in the last cell of the table, which will show the maximum value of 55.

Summary

While the brute force approach guarantees an optimal solution, its exponential time complexity makes it impractical for large problems. The dynamic programming approach, on the other hand, offers a more efficient solution with polynomial time complexity, making it suitable for larger instances, though it requires additional memory for the DP table.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack Problems. Springer.

649 words

Permalink Show parent

Re: Week 5
by Loubna Hussien - Wednesday, 24 July 2024, 11:30 PM

Your explanation of the knapsack problem is detailed and well-structured.
10 words

Permalink Show parent

Re: Week 5
by Prince Ansah Owusu - Thursday, 25 July 2024, 3:21 AM

Your submission provides a thorough and clear explanation of the knapsack problem and effectively contrasts the brute force and dynamic programming approaches. The examples and complexity analysis are well-articulated, making the concepts accessible and easy to understand. Excellent work!
39 words

Permalink Show parent

Re: Week 5
by Liliana Blanco - Wednesday, 24 July 2024, 8:25 PM

The Knapsack Problem (KP), as Krichen and Chaouachi (2014) explain, is a classic issue in combinatorial optimization and computer science. It involves making the most efficient selection from a set of items under certain constraints. The basic concept of the Knapsack Problem is that you are given a set of items, each with a specific value (profit) and weight, along with a knapsack with a limited carrying capacity. The objective is to select a subset of these items to maximize the total value while ensuring that the combined weight does not exceed the knapsack's capacity.

A variant of this issue is the 0-1 Knapsack Problem, in which an item is either completely (1) or completely excluded (0) from the knapsack; partial items are not permitted. A bipartite graph can be used to graphically depict the KP with one set of nodes representing things and the other representing two states—inside the knapsack or outside of it.

The Knapsack Problem (KP) comes in different variations. These variations include the Quadratic KP (QKP), which considers both individual and joint profits for items, the Multiple KP (MKP) which involves multiple knapsacks, each with its capacity, the Multidimensional KP (MDKP) where the weight of items depend on resources, and the Quadratic Multidimensional KP (QMDKP) that combines joint profits and resource-dependent weights with multiple knapsacks (Krichen & Chaouachi, 2014). The KP simulates a range of real-world situations in various industries, such as advertising, budget allocation, telecommunications, cutting stock, freight loading, and financial management.

When dealing with the Knapsack Problem, especially when working it in its 0-1 form, it is essential to contrast two basic algorithm design techniques: dynamic programming and brute force. Every strategy has unique qualities, benefits, and drawbacks.

The brute force approach systematically enumerates all possible combinations of items to find the one that yields the maximum value without exceeding the knapsack's capacity. This method has a time complexity of O(2^n), where n is the number of items, as each item can either be in or out of the knapsack, leading to 2^n possible combinations. Its space complexity is O(1), as it does not require additional space proportional to the input size beyond the storage of the currently evaluated combinations. The brute force approach is simple and guarantees finding the optimal solution since it checks every possibility.

However, its exponential time complexity makes it impractical for large n, and its performance degrades rapidly as the number of items increases.

On the other hand, dynamic programming solves the problem by breaking it down into simpler subproblems, solving each subproblem just once and storing their solutions—usually in a two-dimensional array. The time complexity of this approach is O(nW), where n is the number of items and W is the knapsack capacity, due to the need to compute a solution for each combination of n items and W capacities. The space complexity is also O(nW), as it requires a two-dimensional array to store solutions of subproblems. The dynamic programming approach is more efficient than brute force for large n, with polynomial-time complexity. It guarantees finding the optimal solution and reduces redundant calculations by reusing the solution to subproblems. However, it has higher space complexity, which can be a drawback for very large problems, and it is more complex to understand and implement than the brute force approach.

as an example

Imagine you have a small knapsack that can hold up to 10 pounds. You also have three items to choose from:

1. Item A: Weight 3 pounds, Value $40
2. Item B: Weight 4 pounds, Value $50
3. Item C: Weight 5 pounds, Value $100

Your goal is to select items to maximize the total value without exceeding the 10-pound limit.

In the brute force method, you would check all possible combinations of these items to find the best one.

1. No items: Total weight = 0 pounds, Total value = $0
2. Item A only: Total weight = 3 pounds, Total value = $40
3. Item B only: Total weight = 4 pounds, Total value = $50
4. Item C only: Total weight = 5 pounds, Total value = $100
5. Items A and B: Total weight = 7 pounds, Total value = $90
6. Items A and C: Total weight = 8 pounds, Total value = $140
7. Items B and C: Total weight = 9 pounds, Total value = $150
8. Items A, B, and C: Total weight = 12 pounds (too heavy, not considered)

By checking all combinations, you find that the best choice is to take Items B and C, with a total value of $150 and a total weight of 9 pounds.

In the dynamic programming approach, you would create a table to track the best possible value for each weight capacity from 0 to 10 pounds.

Here's how you would build the table:

Weight. 0. 1. 2. 3. 4. 5 6. 7. 8. 9. 10

Value 0 0 0 40 50 100 90 140 150 150 150

1. For weight 3 (Item A): Maximum value = $40
2. For weight 4 (Item B): Maximum value = $50
3. For weight 5 (Item C): Maximum value = $100
4. For weight 7 (Items A and B): Maximum value = $90
5. For weight 8 (Items A and C): Maximum value = $140
6. For weight 9 (Items B and C): Maximum value = $150

From the table, you can see that the maximum value for the knapsack with a capacity of 10 pounds is $150, achieved by taking Items B and C, which weigh a total of 9 pounds.

The dynamic programming approach, while more complex, is significantly more efficient for larger problems, featuring polynomial time complexity and higher space requirements. The brute force approach, on the other hand, is simple but highly inefficient for large datasets due to its exponential time complexity and minimal space requirements.

References

Krichen, S., & Chaouachi, J. (2014). *Graph-related optimization and decision support systems*. John Wiley & Sons, Incorporated.

1034 words

Permalink Show parent

Re: Week 5

by Loubna Hussien - Wednesday, 24 July 2024, 11:29 PM

Your answer provides a comprehensive and detailed explanation of the Knapsack Problem, along with the brute force and dynamic programming approaches to solving it.

24 words

Permalink Show parent

Re: Week 5

by Mejbaul Mubin - Wednesday, 24 July 2024, 11:59 PM

Hi Liliana Blanco,

You provide a comprehensive and detailed explanation of the Knapsack Problem (KP), effectively breaking down its core concept and various variants. Your inclusion of both brute force and dynamic programming methods to solve the 0-1 Knapsack Problem offers a clear contrast between these approaches, highlighting their respective advantages and disadvantages.

Your explanation is well-structured and informative, especially with the example provided. The step-by-step illustration of both methods makes it easy to understand the practical application of these algorithms. Additionally, your use of a table to demonstrate the dynamic programming approach is particularly effective in visualizing how the optimal solution is achieved.

104 words

Permalink Show parent

Re: Week 5

by Aye Aye Nyein - Wednesday, 24 July 2024, 9:42 PM

Knapsack Problem Overview

The goal of the knapsack problem is to choose a subset of objects with the highest value within a specified weight range. The goal is to maximize the overall value of each item while maintaining the total weight within the knapsack's capacity. Each item has a weight and a value.

Brute Force Approach

The brute force approach involves evaluating all possible combinations of items and selecting the one with the highest value that does not exceed the weight limit of the knapsack.

Advantages:

- Guarantees finding the optimal solution.

Disadvantages:

- Exponential time complexity , where is the number of items.

- Impractical for large due to its time complexity.

Example:

Consider a knapsack with a capacity of 10 and the following items:

- Item 1: weight = 4, value = 6

- Item 2: weight = 3, value = 5

- Item 3: weight = 2, value = 3

To find the optimal subset using brute force:

- Generate all subsets and calculate their total weights and values.

- Compare each subset to determine which one has the highest value without exceeding the weight limit.

Dynamic Programming Approach

Dynamic programming solves the knapsack problem by breaking it down into smaller overlapping subproblems and using a table to store the solutions to these subproblems. This approach allows for efficient computation of the optimal solution without redundant calculations.

Advantages:

- Effective time complexity , where is the knapsack's capacity and is the number of elements.

- Offers the best possible resolution.

Disadvantages:

- Requires additional space to store the DP table, which could be a limitation for very large inputs.

- More complex to implement compared to the brute force approach.

Example:

Using dynamic programming for the same knapsack problem:

Let us create a DP table where represents the maximum value that can be obtained with items up to the -th item and a weight limit .

For the items and knapsack capacity given earlier:

- Initialize for all .

- Fill the DP table based on whether including the current item provides a higher value than excluding it.

After filling the table, will contain the maximum value achievable, which is 14 in this case.

Asymptotic Complexity Analysis

Brute Force:

- Time complexity:

- Space complexity: (ignoring recursive stack space)

Dynamic Programming:

- Time complexity:

- Space complexity:

Conclusion

Dynamic programming offers a significant improvement in efficiency over brute force for solving the knapsack problem, especially for larger instances where and are substantial. While brute force ensures finding the optimal solution, its exponential time complexity makes it impractical for realistic problem sizes. Dynamic programming, with its systematic approach to solving subproblems and leveraging previously computed results, provides a feasible and efficient solution to the knapsack problem.

References:

- Kleinberg, J., & Tardos, É. (2006). Algorithm Design. Pearson Education.

- Skiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer.

542 words

Permalink Show parent

Re: Week 5

by Loubna Hussien - Wednesday, 24 July 2024, 11:27 PM

Your overview of the knapsack problem is thorough and well-organized. You did an excellent job comparing the brute force and dynamic programming approaches, highlighting both their advantages and disadvantages. The inclusion of examples helps to clarify these concepts effectively.

39 words

Permalink Show parent

Re: Week 5

by Prince Ansah Owusu - Thursday, 25 July 2024, 3:19 AM

Your submission provides a clear and concise explanation of the knapsack problem and effectively compares brute force and dynamic programming approaches. The examples and asymptotic complexity analysis are well-presented, making the concepts easy to understand. Great job!

37 words

Permalink Show parent

Re: Week 5

by Loubna Hussien - Wednesday, 24 July 2024, 10:37 PM

Imagine waking up on a mysterious island filled with various valuable items, each with a different value and weight. You have a bag to carry some of these items, but it has a weight limit. Your task is to choose items for your bag in such a way that the total value of the items is maximized (CodesDope, n.d.).

According to GeeksforGeeks (2024), this is commonly known as the knapsack or rucksack problem. We are given NNN items, each with an associated weight and profit, and a bag with a capacity WWW (i.e., the bag can only hold WWW pounds). The goal is to place items in the bag to maximize the total profit without exceeding the bag's capacity. The constraint is that we must either place an item entirely in the bag or not at all (partial items are not allowed).

The knapsack problem appears in various decision-making processes, such as optimizing raw material cuts, selecting investments, securitizing assets, generating keys for Merkle-Hellman, and other tokenization systems using the knapsack concept.

Knapsack Problem Using Brute Force

The brute force approach relies on computing power to test all possible scenarios until the desired outcome is achieved, without enhancing the algorithm's performance. In the context of the knapsack problem, this involves trying all combinations of items that fit in the bag, calculating the total weight and value of each combination, and choosing the combination that maximizes the value without exceeding the weight limit (Gate Vidyalay, n.d.). This method can solve the problem in O(nlogn)O(n \log n)O(nlogn) time due to the time spent organizing elements.

The brute force approach can solve any type of knapsack problem, including those with non-uniform item weights and values, and does not require preprocessing. However, its high time complexity makes it unsuitable for large problem instances (University of the People, n.d.).

Knapsack Problem Using Dynamic Programming

Dynamic Programming (DP) improves upon simple recursion by optimizing recursive solutions that involve repeated calls for the same inputs. By saving the results of subproblems, DP reduces the time complexity from exponential to polynomial (GeeksforGeeks, 2024).

The dynamic programming approach solves the knapsack problem by computing the optimal solution in ascending order, solving smaller subproblems and combining their solutions. It identifies the state (i,w)(i, w)(i,w) with the highest value achievable by selecting a subset of the first iii items with a total weight of www. This involves creating a matrix with items on one axis and potential weights on the other, where each cell represents the highest value achievable for a given weight. By

iteratively updating the matrix, the algorithm finds the optimal set of items to maximize value while staying within the weight limit. This approach has a time complexity of O(NW)O(NW)O(NW), where NNN is the number of items and WWW is the bag's capacity.

The dynamic programming approach is preferred for finding optimal solutions for small and medium-sized problem instances, as it is efficient and fast for minor cases. However, its scalability is limited due to computational expense and large memory requirements, making it unfeasible for large instances of the knapsack problem. The number of unique cases grows exponentially with the number of items, and the large memory needed to store the array adds to its limitations (S et al., n.d.).

In conclusion, while the dynamic programming algorithm is more efficient and optimal for solving the knapsack problem, the brute force algorithm is simpler and more general.

References:

Dope. (n.d.). *Knapsack Programming Using Dynamic Programming and its Analysis*. https://www.codesdope.com/course/algorithms-knapsack-problem/

orGeeks. (2024). 0 1 Knapsack Problem. *GeeksforGeeks*. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Brute force algorithm. (n.d.). https://www-igm.univ-mlv.fr/~lecroq/string/node3.html

sity of the People. (n.d.). *Learning Guide Unit 5*. https://my.uopeople.edu/mod/book/view.php?id=359148&chapterid=426217 \

orGeeks. (2024). Dynamic Programming. *GeeksforGeeks*. https://www.geeksforgeeks.org/dynamic-programming/

gupta, C.H. Papadimitriou, and U.V. Vazirani. (n.d.). *Chapter 6 Dynamic Programming in Algorithms.* http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

625 words

Permalink Show parent

Re: Week 5
by Mejbaul Mubin - Thursday, 25 July 2024, 12:39 AM

Hi Loubna Hussien,

Your post on the knapsack problem provides a thorough overview of both brute force and dynamic programming approaches. You've effectively highlighted the practical applications and trade-offs of each method. The detailed explanation, supported by references, makes the content both informative and accessible. Well done!

47 words

Permalink Show parent

Re: Week 5
by Prince Ansah Owusu - Thursday, 25 July 2024, 3:17 AM

Your submission clearly explains the knapsack problem and effectively compares brute force and dynamic programming approaches, highlighting their advantages and disadvantages. The examples and detailed analysis of asymptotic complexity provide a strong understanding of each methodology. Well done!

38 words

Permalink Show parent

Re: Week 5
by Mahmud Hossain Sushmoy - Thursday, 25 July 2024, 4:05 AM

Hello Loubna,
Your explanation of the knapsack problem and the comparison between brute force and dynamic programming approaches is well-articulated. You've effectively introduced the problem and its significance, highlighting the brute force method's simplicity and its challenges with large datasets, as well as the dynamic programming method's efficiency and its limitations with memory usage and scalability. Thank you for your contribution to the discussion forum this week.
67 words

Permalink Show parent

Re: Week 5
by Liliana Blanco - Thursday, 25 July 2024, 10:36 AM

I really appreciate your clear explanation of the knapsack dilemma and how you made it more accessible by using the island comparison. Comparing the dynamic programming approach to the brute force method was really interesting. I understand that though brute force can solve any knapsack problem, its high time complexity makes it impractical for large-scale examples. On the other hand, dynamic programming requires a lot of memory but provides a more effective solution for small to medium-sized problems. Great work!
80 words

Permalink Show parent

Re: Week 5
by Christopher Mccammon - Wednesday, 24 July 2024, 10:56 PM

The knapsack problem presents a challenge, in optimization where the aim's to maximize the total value of items placed in a knapsack without surpassing its weight limit. Each item comes with a weight and value. The goal is to choose the most valuable combination of items that can fit within the knapsacks capacity. There are two versions; the 0/1 knapsack problem, where each item can be either included or excluded from the knapsack and the fractional knapsack problem, which allows items to be divided permitting portions of an item to be included in the knapsack(Tutorialspoint, n.d).

In terms of finding solutions for the knapsack problem one method involves assessing all combinations of items to identify which combination yields the value while staying under the weight limit. This is known as the brute force algorithm. This approach is advantageous as it guarantees finding a solution through its nature. It also boasts simplicity in comprehension and implementation when dealing with item sets. However when handling sets of items this brute force technique proves inefficient due to its exponential growth in combinations. As a result it becomes computationally expensive and impractical for datasets, with sizes. The time complexity associated with this method is O(2^n) where n represents the number of items – showcasing that for datasets it swiftly becomes unmanageable(Tutorialspoint, n.d).
The dynamic programming technique tackles the knapsack problem by dividing it into subproblems solving each one independently and saving the results to prevent computations. This approach involves creating a table where rows represent items and columns represent weight capacities ranging from 0, to the limit. The table is populated using a formula that determines the value for the few items within a specified weight constraint. One of the benefits of programming is its efficiency, especially when dealing with larger sets of items. It operates with a time complexity of O(nW) where n stands for the number of items and W denotes the weight capacity of the knapsack. However this method necessitates space for storing information in the table, which can become significant, for inputs. Furthermore implementing programming is more intricate compared to using brute force methods (Geeks for Geeks, n. d).
Examples
Consider an example with 3 items having weights [2, 3, 4] and values [3, 4, 5], and a knapsack with a capacity of 5. Using the brute force method, we list all combinations of items:
• (0, 0, 0): weight=0, value=0
• (1, 0, 0): weight=2, value=3
• (0, 1, 0): weight=3, value=4
• (0, 0, 1): weight=4, value=5
• (1, 1, 0): weight=5, value=7
• (1, 0, 1): weight=6, value=8 (exceeds capacity)
• (0, 1, 1): weight=7, value=9 (exceeds capacity)

• (1, 1, 1): weight=9, value=12 (exceeds capacity)
The optimal combination is (1, 1, 0) with weight=5 and value=7(Geeks for Geeks, n. d)..
Using the dynamic programming approach for the same items, we create a DP table initially filled with 0s. The table is filled as follows:
0 1 2 3 4 5
0 0 0 0 0 0 0
1 0 0 3 3 3 3
2 0 0 3 4 4 7
3 0 0 3 4 5 7
The optimal value for a knapsack capacity of 5 is found in cell dp[3][5], which is 7(Geeks for Geeks, n. d)..
Conclusion
When useing the brute force method it's straightforward. Covers all possibilities. It becomes less efficient, for big datasets because of its exponential time complexity. On the side opting for programming offers a more efficient solution, with polynomial time complexity, which works better for larger problems even though it requires more space. Knowing these trade offs is essential when choosing the algorithm to tackle the knapsack problem.
References:
GeeksforGeeks. (n.d.). Dynamic programming. GeeksforGeeks. https://www.geeksforgeeks.org/knapsack-problem-set-2-dp-solution/
Geeks for Geeks.(n.d) Brute-force algorithm. GeeksforGeeks. https://www.geeksforgeeks.org/knapsack-problem-set-1-introduction/
Tutorialspoint. (n.d.). Example of knapsack problem. Tutorialspoint. https://www.tutorialspoint.com/algorithms/knapsack_problem.htm
631 words

Permalink Show parent

Re: Week 5
by Mejbaul Mubin - Thursday, 25 July 2024, 12:35 AM

Hi Christopher McCammon,

Your explanation of the knapsack problem and the comparison between brute force and dynamic programming is clear and concise. You've effectively highlighted the strengths and weaknesses of each method, particularly in terms of efficiency and computational feasibility. Including specific examples and references adds clarity and depth to your analysis. Great job!
54 words

Permalink Show parent

Re: Week 5
by Wingsoflord Ngilazi - Thursday, 25 July 2024, 1:55 AM

You have done to this week's discussion assignment. Keep up the good work.
13 words

Permalink Show parent

Re: Week 5
by Muritala Akinyemi Adewale - Thursday, 25 July 2024, 2:32 AM

Your discussion on the knapsack problem and the comparison between recursive and dynamic programming approaches is very informative. You've done a great job explaining the recursive method's simplicity and the dynamic programming method's efficiency. Including edge cases and real-world examples adds significant value to your analysis. Excellent work!
48 words

Permalink Show parent

Re: Week 5

by Chong-Wei Chiu - Thursday, 25 July 2024, 9:57 AM

Hello, Christopher McCammon. Thank you for sharing your opinion about dynamic programming. You clearly explain the basic concepts of how to solve the knapsack problem and compare the differences between the brute force algorithm and dynamic programming.
37 words

Permalink Show parent

Re: Week 5
by Liliana Blanco - Thursday, 25 July 2024, 10:37 AM

I like how you separated the fractional knapsack problems from the 0/1 knapsack problems. It's interesting how you compare dynamic programming with brute force tactics. I agree that although brute force methods are straightforward and ensure a solution, their exponential time complexity makes them unfeasible for huge datasets. However, although requiring more space, dynamic programming offers a more effective method with polynomial time complexity, making it appropriate for larger situations. Your examples do a good job of explaining how each technique operates. Fantastic work!
84 words

Permalink Show parent

Re: Week 5
by Jerome Bennett - Thursday, 25 July 2024, 12:43 AM

CS 3304 Discussion Forum Unit 5

University of the People

Instructor: Romana Riyaz

The knapsack problem is a classic optimization problem that can be thought of as follows: imagine you have a backpack (knapsack) that can hold a certain weight. You also have a collection of items, each with its own weight and value. The goal is to determine the combination of items to pack in the knapsack so that the total weight doesn't exceed the backpack's capacity, and the total value is maximized.

There are different variants of the knapsack problem, but one of the most common is the "0/1 knapsack problem," where each item can either be taken or left (hence the name 0/1, indicating binary choices) (GeeksforGeeks, 2024).

Brute Force Approach

Brute Force involves evaluating every possible combination of items to find the one with the maximum value without exceeding the weight capacity (June, 2023; Dasgupta et al, n.d.).

Advantages:

1. **Exhaustive Search**: Brute force guarantees finding the optimal solution since every possibility is explored.
2. **Simple Implementation**: It's straightforward to implement, especially for small datasets.

Disadvantages:

1. **Inefficiency**: This approach is highly inefficient for large numbers of items due to the exponential growth in the number of combinations. The time complexity is $O(2^n)$, where n is the number of items. This is because each item has two possibilities (included or not included).
2. **Not Scalable**: The brute force method becomes impractical as the number of items increases.

Dynamic Programming Approach

Dynamic Programming (DP) optimizes the process by breaking the problem down into smaller subproblems and solving each subproblem only once, storing its result for future reference (Dasgupta et al, n.d.). The typical way to implement this is by using a 2D table where the rows represent items and the columns represent the weight capacity from 0 up to the maximum capacity of the knapsack.

Advantages:

1. **Efficiency**: The DP approach significantly reduces the number of subproblems to be solved, bringing the time complexity down to O(n·W), where n is the number of items and W is the capacity of the knapsack (June, 2023).
2. **Optimal Solution**: Like brute force, it guarantees an optimal solution but does so more efficiently.

Disadvantages:

1. **Space Complexity**: The DP approach requires a 2D table to store the results of subproblems, resulting in a space complexity of O(n·W). This can be quite large if the number of items or the knapsack's capacity is large.
2. **More Complex Implementation**: It requires careful handling of indices and can be more challenging to implement correctly.

Example

Let's consider a small example with three items:

- **Item 1**: Weight = 1, Value = 10
- **Item 2**: Weight = 3, Value = 40
- **Item 3**: Weight = 4, Value = 50

Knapsack capacity = 5

Brute Force Solution:

1. Consider all possible combinations of items:

- No items: Total weight = 0, Total value = 0
- Item 1: Total weight = 1, Total value = 10
- Item 2: Total weight = 3, Total value = 40
- Item 3: Total weight = 4, Total value = 50
- Items 1 and 2: Total weight = 4, Total value = 50
- Items 1 and 3: Total weight = 5, Total value = 60
- Items 2 and 3: Total weight = 7, Total value = 90 (exceeds capacity)
- All items: Total weight = 8, Total value = 100 (exceeds capacity)

The best combination that doesn't exceed the capacity is Items 1 and 3, with a total value of 60.
Dynamic Programming Solution:

1. Create a table with 4 rows (including a row for zero items) and 6 columns (for capacities 0 to 5).
2. Fill in the table using the formula:

- If we don't include the item, use the value from the row above.
- If we include the item, take the maximum of the value without the item and the value with the item.

The maximum value at the bottom-right corner of the table gives the optimal solution.
In this example, the DP approach would also find that including Items 1 and 3 gives the maximum value of 60 without exceeding the weight limit.

Reference

Dasgupta, S., Papadimitriou, C.H., Vazirani, U.V. (n.d.). Chapter 6 Dynamic Programming in Algorithms. http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

GeeksforGeeks. (2024, May 30). 01 Knapsack problem. GeeksforGeeks. https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

June, F. (2023, May 22). 0/1 Knapsack Problem - Florian June. Medium. https://medium.com/@florian_algo/0-1-knapsack-problem-eec333f4a991

708 words

Permalink Show parent

Re: Week 5
by Wingsoflord Ngilazi - Thursday, 25 July 2024, 1:48 AM

Well done. Your explanations are very clear. You have provided some relevant examples as well. Keep up the good work.
20 words

Permalink Show parent

Re: Week 5
by Muritala Akinyemi Adewale - Thursday, 25 July 2024, 2:32 AM

Your analysis of the knapsack problem and the contrast between backtracking and dynamic programming methods is impressive. You've effectively demonstrated how backtracking can be intuitive but inefficient for large datasets, while dynamic programming offers a more feasible solution. The detailed examples and thoughtful explanations make your comparison very insightful. Fantastic job!
51 words

Permalink Show parent

Re: Week 5
by Mahmud Hossain Sushmoy - Thursday, 25 July 2024, 4:04 AM

Hello Jerome,
Your discussion on the Knapsack Problem and the comparison of brute force and dynamic programming approaches is comprehensive and well-structured. You've effectively outlined the problem, its variations, and the differences between the two methods. The advantages and disadvantages of each approach are clearly presented, along with their computational complexities. The example provided helps illustrate the application of both methods, making the concepts more tangible. Thank you!
68 words

Permalink Show parent

Re: Week 5
by Jobert Cadiz - Thursday, 25 July 2024, 10:01 AM

Hi Jerome,
The knapsack problem is clearly explained in your response, which also contrasts the dynamic programming and brute force methods. The explanation is clear and precise. The challenge scenario and the objective of optimizing the overall value while staying within the weight limit are precisely depicted. The benefits and drawbacks are clearly stated. It's true that the references to the thorough search guaranteeing an ideal answer and the inefficiency brought on by exponential expansion. Continue your fantastic effort!
79 words

Permalink Show parent

Re: Week 5
by Prince Ansah Owusu - Thursday, 25 July 2024, 2:29 AM

The Knapsack Problem and Algorithmic Approaches

Description of the Knapsack Problem

The Knapsack Problem is a classic optimization problem defined as follows: Given a set of items, each with a specific weight and value, and a knapsack with a maximum capacity, the goal is to select a subset of items that maximizes the total value without exceeding the knapsack's weight capacity. This problem has several variations, including:

1. **0/1 Knapsack Problem**: Each item can either be included in the knapsack or excluded.
2. **Fractional Knapsack Problem**: Items can be broken into fractions, allowing for partial inclusion.

Brute Force Approach

Description: The brute force approach to solving the Knapsack Problem involves evaluating all possible subsets of items to determine which combination provides the maximum value while adhering to the knapsack's capacity (Shaffer, 2011). This method guarantees finding the optimal solution by exhaustively checking every possible combination.

Advantages:

- **Simplicity**: The brute force method is straightforward and conceptually easy to implement (Shaffer, 2011).
- **Exact Solution**: It ensures the optimal solution is found, as every possibility is considered.

Disadvantages:

- **Computational Complexity**: The number of possible subsets is 2n2^n2n, where n is the number of items. This results in an exponential time complexity of O(2n)O(2^n)O(2n), making it impractical for large numbers of items (Dasgupta, Papadimitriou, & Vazirani, 2006).
- **Inefficiency**: Due to its exponential nature, the brute force approach becomes inefficient as the number of items increases.

Example: For a knapsack with a capacity of 50 units and three items:

- Item 1: Weight = 10, Value = 60
- Item 2: Weight = 20, Value = 100
- Item 3: Weight = 30, Value = 120

The brute force method would evaluate all possible combinations of these items, including:

1. No items
2. Item 1 only
3. Item 2 only
4. Item 3 only
5. Item 1 and Item 2
6. Item 1 and Item 3
7. Item 2 and Item 3
8. All three items

The optimal solution is selecting Item 2 and Item 3, which provides a total value of 220 without exceeding the capacity.

Dynamic Programming Approach

Description: Dynamic Programming (DP) solves the Knapsack Problem by breaking it down into simpler overlapping sub-problems and storing the results of these sub-problems to avoid redundant computations (Dasgupta, Papadimitriou, & Vazirani, 2006). The approach uses a table to keep track of the maximum value achievable for each sub-capacity and item combination.

Advantages:

- **Efficiency**: DP reduces the number of computations by storing intermediate results, resulting in a time complexity of O(n×W)O(n \times W)O(n×W), where n is the number of items and W is the knapsack's capacity (Shaffer, 2011).
- **Optimal Solution**: It guarantees finding the optimal solution similar to brute force but in a more efficient manner.

Disadvantages:

- **Space Complexity**: The DP table requires O(n×W)O(n \times W)O(n×W) space, which can be substantial if the capacity W is large (Shaffer, 2011).
- **Implementation Complexity**: Implementing DP requires careful management of state transitions and table updates, which can be more complex compared to brute force.

Example: Using the same knapsack and items as above:

1. Create a DP table where dp[i][w] represents the maximum value achievable with the first i items and a knapsack capacity of w.
2. Initialize the table with zero values.
3. Fill the table by iterating through each item and capacity, updating values based on whether including an item improves the maximum value.

The final value at dp[n][W] (where n is the number of items and W is the capacity) will indicate the maximum value achievable. For the given example, the DP table shows that the maximum value achievable is 220 with the items selected being Item 2 and Item 3.

Asymptotic Complexity

- **Brute Force**: Time Complexity O(2n)O(2^n)O(2n), Space Complexity O(1)O(1)O(1) (Shaffer, 2011).
- **Dynamic Programming**: Time Complexity O(n×W)O(n \times W)O(n×W), Space Complexity (O(n \times W)\ (Dasgupta, Papadimitriou, & Vazirani, 2006).

References

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006). *Algorithms*. Retrieved from Dynamic Programming Chapter

Shaffer, C. A. (2011). *A Practical Introduction to Data Structures and Algorithm Analysis*. Blacksburg, VA: Virginia Tech University.

Leiserson, C. E. (n.d.). *Dynamic Programming*. Retrieved from MIT Lecture

663 words

Permalink Show parent

Re: Week 5
by Muritala Akinyemi Adewale - Thursday, 25 July 2024, 2:31 AM

Your explanation of the knapsack problem and the distinction between greedy algorithms and dynamic programming is excellent. You've clearly outlined the scenarios where each method excels, making it easier to understand their practical applications. The step-by-step walkthroughs and illustrative examples greatly enhance the clarity of your presentation. Well done!
49 words

Permalink Show parent

Re: Week 5
by Mahmud Hossain Sushmoy - Thursday, 25 July 2024, 4:03 AM

Hello Prince,
Your description of the Knapsack Problem and the comparison of brute force and dynamic programming approaches is clear and well-organized. You effectively outline the problem and its variations, while detailing the advantages and disadvantages of each algorithmic approach. Your explanation of the brute force method and its exponential complexity highlights its impracticality for large datasets, whereas the dynamic programming approach is well-explained with its efficient time complexity and the trade-off of higher space complexity. The examples provided for both methods illustrate the concepts effectively. Thank you for your contribution!
91 words

Permalink Show parent

Re: Week 5
by Mejbaul Mubin - Thursday, 25 July 2024, 8:25 AM

Hi Prince Ansah Owusu,

You provide a clear and comprehensive explanation of the Knapsack Problem, covering both brute force and dynamic programming approaches. your explanation is well-structured and informative.
29 words

Permalink Show parent

Re: Week 5
by Liliana Blanco - Thursday, 25 July 2024, 10:40 AM

You have provided a detailed and lucid explanation of the knapsack dilemma. I like how you compared and contrasted the benefits and drawbacks of the dynamic and brute force programming techniques. Your three-item example clearly demonstrates how both approaches function. I agree that although brute force methods ensure a precise result, their exponential time complexity makes them unfeasible for larger datasets. However, although requiring more space, dynamic programming provides a more effective approach with polynomial time complexity.
77 words

Permalink Show parent

Re: Week 5
by Mahmud Hossain Sushmoy - Thursday, 25 July 2024, 3:40 AM

The Knapsack Problem involves selecting items from a set, each with a specific weight and value, to place in a knapsack with a limited weight capacity. The objective is to maximize the total value of items in the knapsack without exceeding its weight limit (GeeksforGeeks, 2024). This problem requires determining which items to include to achieve the highest possible total value.

The brute force method exhaustively generates and evaluates all possible combinations of items. While this approach guarantees finding the optimal solution by considering every possibility, it suffers from severe inefficiency. As the number of items increases, the number of subsets to evaluate grows exponentially, resulting in a time complexity of O(2^n) and a space complexity of O(n). Consequently, this method becomes impractical for large datasets due to its prohibitive computational demands.

Dynamic programming (DP) provides a more efficient method by dividing the problem into smaller, more manageable subproblems. It utilizes a table to store intermediate results, recording the maximum achievable value for each weight limit up to the knapsack's capacity. This method ensures that each subproblem is solved only once, significantly reducing redundant calculations. The DP approach boasts a time complexity of O(nW) and a space complexity of O(nW), where n represents the number of items and W the knapsack's weight capacity (University of the People, n.d.). While it requires additional memory for the DP table, this method is far more efficient for larger inputs.

The brute force approach, while straightforward and guaranteed to find the optimal solution, is rendered impractical for large-scale problems due to its exponential time complexity. In contrast, the dynamic programming method provides a more efficient solution, making it suitable for larger datasets despite its increased space requirements. This trade-off between time and space efficiency often favors the DP approach in real-world applications.

Consider a scenario with items weighing {1, 2, 3} units and valued at {6, 10, 12} respectively, with a knapsack capacity of 5 units. The brute force approach would evaluate all possible item combinations, while the DP method constructs a table to efficiently determine the optimal subset. Employing the DP strategy reveals that the maximum attainable value without exceeding the weight limit is 22, achieved by selecting the items weighing 2 and 3 units.

References

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (n.d.). *Algorithms*. Retrieved from http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf

GeeksforGeeks. (2024, July 6). *Introduction to knapsack problem, its types, and how to solve them*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

University of the People. (n.d.). *UNIT 3: Optional Video Lectures*. Retrieved from https://my.uopeople.edu/mod/folder/view.php?id=423550

417 words

Permalink Show parent

Re: Week 5
by Winston Anderson - Thursday, 25 July 2024, 4:17 AM

Hi Mahmud,
Your explanation of the knapsack problem and the comparison between the brute force and dynamic programming (DP) methods is clear and informative. Overall, the content is well-structured and provides a solid foundation for understanding the knapsack problem and its solutions.

42 words

Permalink Show parent

Re: Week 5
by Chong-Wei Chiu - Thursday, 25 July 2024, 5:20 AM

0 words

Permalink Show parent

Re: Week 5
by Chong-Wei Chiu - Thursday, 25 July 2024, 5:31 AM

Reference:

Schaffer, C.A. (2011). A Practical Introduction to Data Structures and Algorithms Analysis (3.1 ed.). Blacksburg, VA: Virginia Tech University, Department of Computer Science. Available at http://people.cs.vt.edu/~shaffer/Book/C++3e20100119.pdf

Dasgupta, S., Papadimitriou, C.H., & Vazirani, U.V. (2006). Algorithms. Berkeley, CA: University of California Berkeley, Computer Science Division. Available at http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf

48 words

Permalink Show parent

Re: Week 5
by Christopher Mccammon - Thursday, 25 July 2024, 10:10 AM

H Chiu,
You have provided a comprehensive analysis and demonstrated a solid understanding of the topic.

16 words

Permalink Show parent

Re: Week 5

by Natalie Tyson - Thursday, 25 July 2024, 6:09 AM

We will be going over brute force and dynamic programming algorithms to detail the differences between these two methods for approaching algorithms. I will also describe the knapsack problem in my own words.

Brute force programming:
Not perhaps the most optimal approach for solving algorithms because this method is not flexible, but it generally gets the job done. We have a tendency to solve all solutions to get the answers by testing all of the different routing combinations. If there are a large set of problems to get through and solve, some of the better matched algorithms used to problem solve get results much more quickly because they do not try to solve all of the problems in a way that would be efficient, just the same method over and over again.

Dynamic Programming:
Where the problem is broken down into smaller subproblems to solve using optimal approaches for the algorithms. Once the program has solved the smaller sub-problems, their solutions are overlapped against each other and compared for similarity instead of being solved in and of themselves.

Differences between these programming approaches:

Brute Force programming will evaluate all possible outcomes for a given problem while the dynamic programming avoids recursion and storing subproblem solutions to find all possible outcomes. There are less iterations in dynamic programming than there would be with brute force. The dynamic programming is simply much more efficient for larger sets of problems and will save on time and computations. If storage capacity is limited, brute force programming generally requires a much smaller amount of space for storage because it does not need the extra space for sub-problem solutions to be stored.

The **knapsack problem** is a problem that many of us have heard of by this point in our early math classes. There are a few types of knapsack problems, but typically you are given a problem where you have a bag that you are holding with a weight capacity assigned to it. You have a set of items with different weights and values associated with them. You need to calculate what the max total value could be that you fill your bag with, assuming the total weight is less than what the bag's capacity is.

The brute force approach to the knapsack problem:

It would evaluate all outcomes, it would have an exponential complexity of O (2^n). Brute force requires less space than DP. It has constant space O(1). It would calculate an optimal solution. It would also be easier to implement than a DP would, but if you get larger instances it would be less efficient.

Dynamic approach to knapsack problem:

The time complexity for DP is O(nW), making it optimal for larger instances. DP requires more space for storage than Brute force programming would. It is also guaranteed to calculate an optimal solution. Requires quite a bit more understanding to conceptualize the subproblem solutions and make the DP approach efficient.

Conclusion

Both brute force and dynamic programming are capable of solving the knapsack problems optimally. If you have a lot of small instances then brute force would be a better algorithm for approaching those problems, but for larger problems using a dynamic program would be more time and space effective, even with its higher capacity for data storage required to store its subproblem solutions.

Citations

- GeeksforGeeks. (2024a, March 26). Difference between brute force and Dynamic Programming. https://www.geeksforgeeks.org/difference-between-brute-force-and-dynamic-programming/#

- GeeksforGeeks. (2024b, July 6). Introduction to knapsack problem, its types and how to solve them. https://www.geeksforgeeks.org/introduction-to-knapsack-problem-its-types-and-how-to-solve-them/

581 words

Permalink Show parent

Re: Week 5
by Jobert Cadiz - Thursday, 25 July 2024, 9:58 AM

Hi Natalie,
Your answer clearly identifies the knapsack problem and mentions its typical form involving a weight and values. It attempts to compare brute force and dynamic programming approaches, outlining some of their advantages and disadvantages. The answer would benefit from concrete examples illustrating both the brute force and dynamic programming methods. It would help in understanding how each algorithm works in practice, unfortunately, it is not present in your post
71 words

Permalink Show parent

Re: Week 5
by Tamaneenah Kazeem - Thursday, 25 July 2024, 10:54 AM

Excellent post Natalie. Keep it up
6 words

Permalink Show parent

Re: Week 5
by Tamaneenah Kazeem - Thursday, 25 July 2024, 6:21 AM

In your own words describe the 'knapsack problem'. Further, compare and contrast the use of a brute force and a dynamic programming algorithm to solve this problem in terms of the advantage and disadvantages of each. An analysis of the asymptotic complexity of each is required.

The knapsack problem is a greedy algorithm method in which the goal is to maximize the total value of item that can fit in a knapsack without exceeding the weight capacity. One thing about the items is that they have specific weights and values, So, the challenge there is to decide which will be chosen in order to achieve the maximum value without surpassing the weight limit of the knapsack.

The brute-force approach works by solving all possible combinations of items that can fit in the knapsack. For each combination, we calculate the total weight and value and then check if it fits the requirement; it satisfies the weight constraint while maximizing the value.

An advantage of this approach is that it ensures the optimal solution is as it exhaustively explores all possible combinations. This method is also straightforward and simple.

However, it does have its disadvantages. It can be very inefficient when it comes to large numbers because the asymptotic complexity makes the number of combinations grow exponentially with the number of items; O (2^n).

The other approach is called a dynamic programming approach. This method works by breaking the problem at hand into smaller subproblems. The solutions to these are then stored so as to avoid meaningless computations. The dynamic approach usually involves computing a table in which each entry represents the maximum value that can be achieved with a subset of items and a specific weight capacity.

An advantage of the dynamic programming approach is that it can be very useful for problems like the knapsack problem, that is, problems with overlapping subproblems. This is because it can prevent recalculations to the same subproblems. Another good property is that it provides an optimal solution with a time complexity that is polynomial relative to the number of items and the capacity of the knapsack.

Like all things however, it also has some disadvantages. It requires more memory space compared to the brute force approach due to the need to store solutions to subproblems. Also, implementing it may not be too friendly to beginners as it requires understanding of the relationship between subproblems and how to construct the solution table.

It has a polynomial time complexity, O (n * W): where n is the number of items and W is the capacity of the knapsack. This makes it feasible for large inputs compared to the exponential time complexity of brute force.

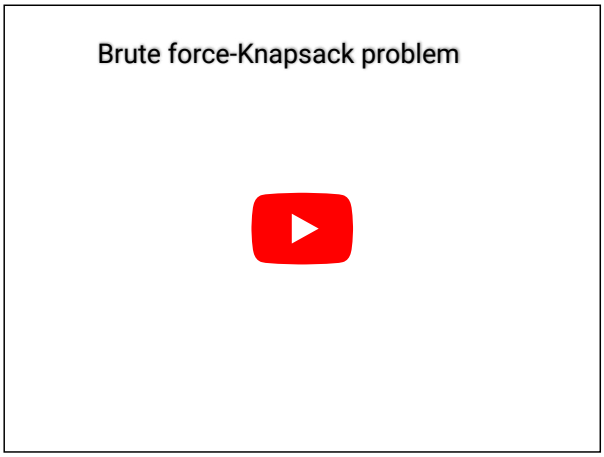
Now we'll compare in terms of efficiency, space, and optimality:

- **Efficiency:** Dynamic programming is significantly more efficient than brute force for large inputs due to its polynomial time complexity.
- **Space:** Brute force requires less memory overhead compared to dynamic programming, which maintains a table of solutions.
- **Optimality:** Both approaches can find the optimal solution, but brute force guarantees it by checking all combinations, while dynamic programming achieves optimality through systematic computation and memorization.

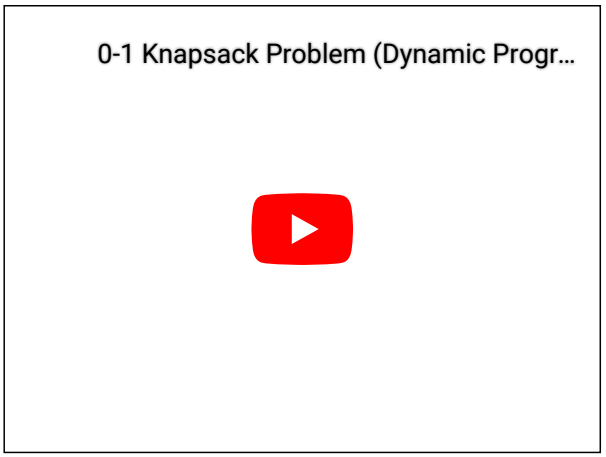
To conclude, although using the brute force approach can be simple and guarantee an optimal solution, its exponential time complexity makes it impractical for large instances. Dynamic programming, on the other hand, sacrifices some memory space but provides a scalable solution with polynomial time complexity, making it suitable for practical applications of the knapsack problem with larger inputs.

References:

Priya, S. (2021, June 15). *Brute Force-Knapsack problem*. YouTube.



Dojo, CS. (2016, March 13). *0-1 Knapsack Problem (dynamic programming)*. YouTube.



Wang, J. (2023, September 2). *What is the knapsack problem? #shorts*. YouTube. https://youtube.com/shorts/7wg3fk74W_Q?si=a5c-w8pIvPbHNDrP

Chapter 6 Dynamic Programming in Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani available at http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

632 words

Permalink Show parent

Re: Week 5
by Jobert Cadiz - Thursday, 25 July 2024, 9:35 AM

Hi Tamaneenah,
Your introduction to the knapsack problem is clear and your analysis of the asymptotic complexity is correct. You cover the advantages and disadvantages well, but you could also mention that the brute-force method is easier to implement for those who are not familiar with dynamic programming. Including specific examples to illustrate both approaches would enhance understanding.
58 words

Permalink Show parent

Re: Week 5
by Tamaneenah Kazeem - Thursday, 25 July 2024, 10:53 AM

Thanks for the feedback. Appreciate it.
6 words

Permalink Show parent

Re: Week 5
by Jobert Cadiz - Thursday, 25 July 2024, 6:45 AM

Discussion Forum Unit 5

The Knapsack Problem

The knapsack problem is like a thief who must decide what to take from a pile of loot, given that his bag ("knapsack") can only hold a certain maximum weight *W*. Each item has a weight w_i and a value v_i. The goal is to find the most valuable combination of items that fit within the weight limit.

Two main versions of the problem are considered (considering the definition above):

1. **Knapsack with repetition**: The thief can take an unlimited number of each item.

2. **Knapsack without repetition**: The thief can take only one of each item.

Brute Force vs. Dynamic Programming

Brute Force Approach

The brute force approach involves generating all possible combinations of items and selecting the combination with the highest value that does not exceed the knapsack's weight limit.

Advantages:

- Simple to implement.
- Finds the optimal solution because it checks all combinations.

Disadvantages:

- Extremely inefficient for large inputs due to exponential growth in the number of combinations.
- Asymptotic complexity is $O(2^n)$, where n is the number of items. This makes the brute force method impractical for large n.

Dynamic Programming Approach

Dynamic programming (DP) provides a more efficient way to solve the knapsack problem by breaking it down into smaller subproblems and storing the results to avoid redundant calculations.

Advantages:

- More efficient than brute force, especially for large inputs.
- Asymptotic complexity is O(nW), where n is the number of items and W is the maximum weight capacity of the knapsack.

Disadvantages:

- Requires additional memory to store intermediate results.
- Can still be slow if W is very large, but manageable compared to brute force.

These examples were taken from the book Algorithms by Dasgupta et al. (2006):

Knapsack with Repetition

Suppose the knapsack capacity W=10 and we have the following items:

We use dynamic programming to define K(w) as the maximum value we can achieve with a knapsack of capacity www. The formula is:

We start with K (0) = 0 and calculate for each w from 1 to W:

1. K (0) = 0
2. For w=1 to W:

- K (1) = 0 (no items can fit)
- K (2) = 9 (item 4)
- K (3) = 14 (item 2)
- K (4) = 18 (two of item 4)
- K (5) = 18 (item 2 and item 4)
- K (6) = 30 (item 1)
- K (7) = 30 (item 1 and item 4)
- K (8) = 36 (item 1 and two of item 4)

- K (9) = 42 (item 1 and item 2)
- K (10) = 48 (item 1 and two of item 4)

The maximum value with a capacity of 10 and allowing repetition is **$48.**

Knapsack without Repetition

Using the same items, we define K (w, j) as the maximum value we can achieve with a knapsack of capacity w using the first j items. The formula is:

We start with K (w, 0) = 0 and K (0, j) = 0. We fill the table as follows:

1. Initialize K (w, 0) = 0 and K (0, j) = 0.
2. For j =1 to n:

For w = 1 to W:
- If $w_j > w$: K (w, j) = K (w, j − 1)
- Else: K (w, j) = max {K (w, j − 1), K (w − w_j, j−1) + v_j}

We fill the table:

The maximum value with a capacity of 10 without allowing repetition is **$46.**

Analysis

Brute Force:

- Time complexity: $O(2^n)$
- Very slow and impractical for large n.

Dynamic Programming:

- Time complexity: O(nW)
- Much faster and more practical for larger n and W.

Conclusion

The knapsack problem can be solved much more quickly and effectively with dynamic programming than with the brute force approach. Larger inputs can be handled by it because of its huge computing time reduction, even if it requires more memory.

References

Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2006, July 18). Algorithms: Chapter 6 – Dynamic programming Algorithms. https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf

654 words

Permalink Show parent

Re: Week 5

by Christopher Mccammon - Thursday, 25 July 2024, 10:21 AM

Hi Cadiz,
You have provided a comprehensive analysis and demonstrated a solid understanding of the topic.
16 words

Permalink Show parent

Re: Week 5

by Tamaneenah Kazeem - Thursday, 25 July 2024, 10:53 AM

Well done Jobert. It seems to me you have fulfilled the requirements of this assignment. Good job.

17 words

Permalink Show parent