Thief with a Knapsack,

A Series of Crimes!

0. Abstract

Knapsack problem is a classic optimization problem that involves selecting a subset of items with maximum value while keeping the total weight of the items within a certain limit. In this paper, I provide a comprehensive overview of the Knapsack problem, including its significance and variations such as the Zero-One, Complete, and Multiple Knapsack problems. We delve into the common solution methods for the Zero-One Knapsack problem, with a particular emphasis on the dynamic programming approach. Finally, I present case studies highlighting the application of Knapsack problem techniques to Leetcode computational problems such as partition equal subset sum, coin change, target sum, and ones and zeros.

1. Introduction

Imagine you're a thief robbing a museum exhibit of tantalizing jewelry, geodes and rare gems. You're new at this, so you only brought a single backpack. Your goal should be to get away with the most valuable objects without overloading your bag until it breaks or becomes too heavy to carry. How do you choose among the objects to maximize your loot? You could list all the artifacts and their weights to work out the answer by hand. But the more objects there are, the more taxing this calculation becomes for a person—or a computer.

This fictional dilemma, the "knapsack problem," belongs to a class of mathematical problems famous for pushing the limits of computing. And the knapsack problem is more than a thought experiment. "A lot of problems we face in life, be it business, finance, including logistics, container ship loading, aircraft loading — these are all knapsack problems," says Carsten Murawski, professor at the University of Melbourne in Australia. "From a practical perspective, the knapsack problem is ubiquitous in everyday life."

The knapsack problem belongs to a class of "NP" problems, which stands for "nondeterministic polynomial time." The name references how these problems force a computer to go through many steps to arrive at a solution, and the number increases dramatically based on the size of the inputs—for example, the inventory of items to choose from when stuffing a particular knapsack. By definition, NP problems also have solutions that are easy to verify (it would be trivial to check that a particular list of items does, in fact, fit in a backpack).

2. 0/1 Knapsack Problem

a. Problem Definition

Remember the thief example we mentioned earlier? The problem can be simplified as follows: you have a backpack with a total capacity of ***c***, and there are ***n*** items to choose from. Each item has its own weight ***w*** and value ***v***, and you need to find the appropriate combination to maximize the total value in your backpack.

But why do we call it the 0/1 Knapsack Problem?

It's because each item is unique, and there are only two options for each item: take it or leave it. We denote taking an item as 1 and leaving it as 0. As we can't split an item, taking only half of it is not allowed. This is why it's called the 0/1 Knapsack problem.

Let's illustrate the problem with a more specific example.

Suppose our backpack has a capacity of 10, and there are 4 items in the museum, with the following values and weights:

We use Xi to represent the choice of the i th treasure; Xi = 1 means we have chosen the treasure, and Xi = 0 means we have not. Vi represents the value of the i th treasure, and Wi represents the weight of the i th treasure.

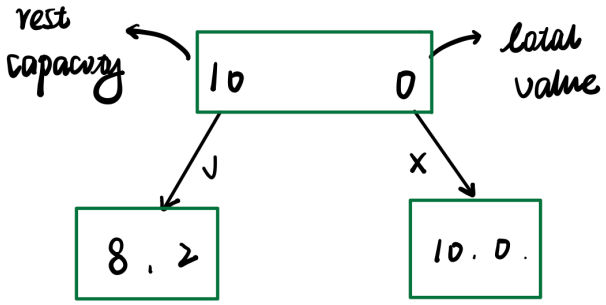
item	1	2	3	4
value	2	4	3	7
weight	2	3	5	5

capacity: 10 n: 4.

Naturally, we can think of the following constraints:

X1W1 + X2W2 +… + XnWn <= Capacity

The initial state of the backpack has a capacity of 10 and a total value of 0. For the first treasure, the current capacity is 10, more than enough to accommodate its weight, so there are two choices: take it or leave it. When we choose a treasure, the backpack's capacity decreases, but the total value of items inside increases, as shown below:



Each choice we make splits each possibility into two. The subsequent three choices work the same way. Finally, we find the maximum value among these outcomes, which is 13 in this case.

b. Common Approaches

Of course, it's not feasible to draw diagrams every time to solve the problem. Here are three common methods for solving the Knapsack problem.

b.1. Brute Force

The brute force method generates all possible subsets of items and computes the total value and weight for each subset. It then selects the subset with the highest value that does not exceed the knapsack's capacity.

```python
from itertools import chain, combinations

items = [("Item1", 2, 2), ("Item2", 4, 3), ("Item3", 3, 5), ("Item4", 7, 5)]
capacity = 10

def brute_force_knapsack(items, capacity):
    def total_value(subset):
        return sum(item[2] for item in subset)

    def total_weight(subset):
        return sum(item[1] for item in subset)

    subsets = chain.from_iterable(combinations(items, r) for r in range(len(items)+1))
    valid_subsets = (s for s in subsets if total_weight(s) <= capacity)
    best_subset = max(valid_subsets, key=total_value)

    return best_subset, total_value(best_subset)

solution, value = brute_force_knapsack(items, capacity)
print(solution, value)
```

b.2 Greedy Algorithm

The greedy algorithm sorts the items based on their value-to-weight ratio, and iteratively adds the item with the highest ratio to the knapsack until the capacity is reached or no more items can be added. (Picture_1 in next Page)

b.3 Dynamic Programming

The dynamic programming approach constructs a table with dimensions (number of items + 1) x (capacity + 1), and iteratively computes the optimal solution for each subproblem. The final solution can be found in the last cell of the table. (Picture_2 in next Page)

```python
def greedy_knapsack(items, capacity):
    sorted_items = sorted(items, key=lambda x: x[2]/x[1], reverse=True)
    total_value = 0
    solution = []

    for item in sorted_items:
        _, weight, value = item
        if capacity >= weight:
            solution.append(item)
            capacity -= weight
            total_value += value

    return solution, total_value

solution, value = greedy_knapsack(items, capacity)
print(solution, value)
```

Picture_1

```python
def dynamic_programming_knapsack(items, capacity):
    n = len(items)
    dp_table = [[0] * (capacity + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        _, item_weight, item_value = items[i - 1]
        for w in range(capacity + 1):
            if item_weight <= w:
                dp_table[i][w] = max(dp_table[i - 1][w], dp_table[i - 1][w - item_weight] + item_value)
            else:
                dp_table[i][w] = dp_table[i - 1][w]

    solution = []
    w = capacity
    for i in range(n, 0, -1):
```

Picture_2

c. Comparative Analysis of Approaches

1. Brute Force:

→ **Complexity**: This approach examines all possible subsets of items, so its time complexity is O(2^n), where n is the number of items. It is computationally expensive and becomes impractical for large instances of the problem.

→ **Optimality**: The brute force method guarantees an optimal solution, as it exhaustively checks all possibilities.

→ **Implementation**: The implementation of the brute force method is straightforward using standard Python libraries.

2. Greedy Algorithm:

→ **Complexity:** The time complexity of the greedy algorithm is O(n log n) due to sorting the items based on value-to-weight ratio, where n is the number of items. It is much faster

than the brute force method.

→ **Optimality:** The greedy algorithm does not guarantee an optimal solution for the 0/1 Knapsack problem, as it focuses on immediate value without considering the overall combination of items.

→ **Implementation**: The greedy algorithm is easy to implement and requires only a single pass through the sorted items.

3. Dynamic Programming:

→ **Complexity:** The time complexity of dynamic programming for the Knapsack problem is O(nW), where n is the number of items and W is the capacity of the knapsack. This is generally more efficient than the brute force approach, especially when the capacity is relatively small compared to the number of items.

→ **Optimality:** DP guarantees an optimal solution for the 0/1 Knapsack problem.

→ **Implementation:** Implementing dynamic programming is more complex than the greedy algorithm, but less complex than branch and bound. It requires constructing and manipulating a two-dimensional table for memoization, and then reconstructing the solution from the table.

In summary, each method has its trade-offs in terms of computational complexity, optimality, and ease of implementation. The brute force method guarantees an optimal solution but is computationally expensive. The greedy algorithm is faster and easier to implement but may not always find the optimal solution. Dynamic programming provides a balance between complexity and optimality, offering an efficient solution for smaller instances of the problem.

3. In-depth Analysis of DP Approach

a. Get Recursion Function

Let's analyze how to extend the above specific example to the general case. To achieve this, we need to abstract the problem and build a model, then divide it into smaller subproblems and find the recursive relationship.

1. Abstract the problem: The Knapsack problem can be abstracted as finding a combination. Xi takes 0 or 1, indicating whether the i th item is taken or not. Vi represents the value of the i th item, Wi represents the weight of the i th item, the total number of items is n, and the backpack capacity is c.
2. Model the problem: Find the maximum value of (X1v1 + X2V2 + X3V3 + ... + XnVn)
3. Constraints: X1W1 + X2W2 + X3W3 + ... + XnWn ≤ c
4. Define the function KS(i, j): representing the value of the best combination of the first i items when the remaining capacity of the backpack is j.

What is the recursive relationship here? For the i th item, there are two possibilities:

The remaining capacity of the backpack is insufficient to accommodate the item; in this case, the value of the backpack is the same as the value of the first i-1 items, KS(i, j) = KS(i-1, j).

The remaining capacity of the backpack can hold the item; at this point, a judgment is needed because taking the item does not necessarily lead to the maximum value of the final combination. If the item is not taken, the value is KS(i-1, j). If the item is taken, the value is KS(i-1, j-wi) + vi. The recursive relationship is obtained by choosing the larger value between the two:

$$KS(i, j) = KS(i-1, j) \quad (j < Wi)$$

$$KS(i, j) = \max\{KS(i-1, j), KS(i-1, j-Wi) + Vi\} \quad (j>Wi)$$

For the subproblems of this issue, it is necessary to provide a detailed explanation. The original problem is to put n items into a backpack with a capacity of c. The subproblem is to put the first i items into a backpack with a capacity of j, and the optimal value obtained is KS(i, j). Considering only whether to place the ith item, the problem can be transformed into one involving only the first i-1 items. If the ith item is not placed, the problem becomes "the optimal value combination of placing the first i-1 items in a backpack with a capacity of j", with the corresponding value KS(i-1, j). If the ith item is placed, the problem becomes "the optimal value combination of placing the first i-1 items in a backpack with a capacity of j-wi", with the corresponding value KS(i-1, j-wi) + vi.

Therefore, the recursive solution can be easily derived:

```python
class Solution:
    def __init__(self):
        self.vs = [0, 2, 4, 3, 7]
        self.ws = [0, 2, 3, 5, 5]

    def test_knapsack1(self):
        result = self.ks(4, 10)
        print(result)

    def ks(self, i, c):
        result = 0
        if i == 0 or c == 0:
            result = 0
        elif self.ws[i] > c:
            result = self.ks(i-1, c)
        else:
            tmp1 = self.ks(i-1, c)
            tmp2 = self.ks(i-1, c-self.ws[i]) + self.vs[i]
            result = max(tmp1, tmp2)
        return result
```

b. Top-down and Bottom-up Approaches

The difference between the top-down approach and the divide-and-conquer approach is the addition of an array to store intermediate results to reduce repeated calculations. Here, we only need to define an additional two-dimensional array.

i \ j	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											
4											

In the table, each cell represents a subproblem, and our final problem is to find the value in the bottom-right corner, which is the value when i=4 and j=10. Here, our initial condition is that the KS value is 0 when i=0 or j=0. This is easy to understand: if there are no available items or if the remaining capacity is 0, then the maximum value is naturally 0. The code is as follows:

```python
class Solution:
    def __init__(self):
        self.vs = [0, 2, 4, 3, 7]
        self.ws = [0, 2, 3, 5, 5]
        self.results = [[None] * 11 for _ in range(5)]

    def test_knapsack2(self):
        result = self.ks2(4, 10)
        print(result)

    def ks2(self, i, c):
        result = 0
        # If the result has already been calculated, return it directly
        if self.results[i][c] is not None:
            return self.results[i][c]
        if i == 0 or c == 0:
            # Base case
            result = 0
        elif self.ws[i] > c:
            # The item cannot be packed
            result = self.ks2(i-1, c)
        else:
            # The item can be packed
            tmp1 = self.ks2(i-1, c)
            tmp2 = self.ks2(i-1, c-self.ws[i]) + self.vs[i]
            result = max(tmp1, tmp2)
        self.results[i][c] = result
        return result
```

Next, let's use the bottom-up approach to solve this problem. The idea is simple: we just need to keep filling in the table. Continuing with the table above, we start filling it in row by row.

We'll skip the process of inserting the first item and directly proceed to when i=2, where there are two items to choose from. At this point, we can apply the recursive formula above to make the decision. Here, we'll analyze when i=2 and j=3:

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2
2	0	0	2	6							
3	0										
4	0										

$KS(2,3) = \max\{KS(1,3), KS(1,3-3)+4\} = 4$

$KS(1,3) = 2$, $KS(1,3-3) = 0$

Fill in the remaining cells using the same method:

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2
2	0	0	2	6	6	6	6	6	6	6	6
3	0	0	2	6	4	6	6	6	7	7	9
4	0	0	2	6	4	7	7	9	11	11	13

Now, we have our final result: 13. Based on this result, we can backtrack to find the selection of each item. The search method is simple: start from i=4 and j=10. If KS(i-1,j) = ks(i,j), it means that the i th item has not been selected; continue searching from KS(i-1,j). Otherwise, the i th item has been selected, and we start searching from KS(i-1,j-wi).

i \ j	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	2	2	2	2	2	2	2	2
2	0	0	2	6	6	6	6	6	6	6	6
3	0	0	2	6	4	6	6	6	7	7	9
4	0	0	2	6	4	7	7	9	11	11	13

The complete code is as follows:

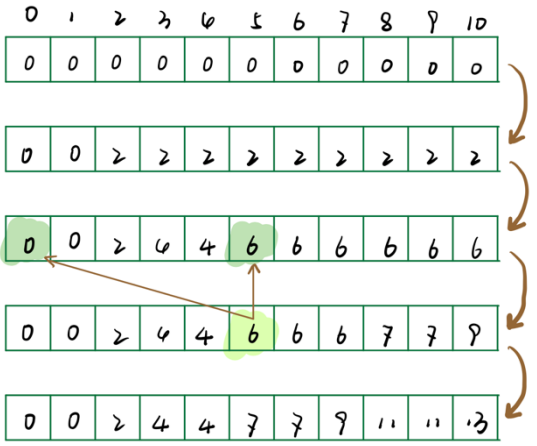
```python
class Solution:
    def __init__(self):
        self.vs = [0, 2, 4, 3, 7]
        self.ws = [0, 2, 3, 5, 5]
        self.results = [[None] * 11 for _ in range(5)]

    def test_knapsack3(self):
        result = self.ks3(4, 10)
        print(result)

    def ks3(self, i, j):
        # Initialize the first row and the first column
        for m in range(i+1):
            self.results[m][0] = 0
        for m in range(j+1):
            self.results[0][m] = 0
        # Start filling in the table
        for m in range(1, i+1):
            for n in range(1, j+1):
                if n < self.ws[m]:
                    # The item cannot be packed
                    self.results[m][n] = self.results[m-1][n]
                else:
                    # The item can be packed
                    if self.results[m-1][n] > self.results[m-1][n-self.ws[m]] + self.vs[m]:
                        # Do not pack the item
                        self.results[m][n] = self.results[m-1][n]
                    else:
                        # Pack the item
                        self.results[m][n] = self.results[m-1][n-self.ws[m]] + self.vs[m]
        return self.results[i][j]
```

c. Optimizations and Trade-offs

We can see that solving KS(i,j) each time is only related to KS(i-1,m) {m:1...j}. In other words, if we know K(i-1,1...j), we can definitely find KS(i,j). To better understand this, let's draw another illustration:



The next layer can find the answer based on the results of the previous layer. For example, when i=3 and j=5, when solving this subproblem's optimal solution, according to the derivation formula above, KS(3,5) = max{KS(2,5), KS(2,0) + 3} = max{6,3} = 6; If we have found the solutions to all subproblems when i=2, then it's easy to find the solutions to all subproblems when i=3.

Therefore, we can optimize the solution space by compressing the two-dimensional array into a one-dimensional array. In this case, the state transition equation becomes:

$$KS(j) = max\{KS(j), KS(j - Wi) + Vi\}$$

Here, KS(j - wi) is equivalent to the original KS(i-1, j - wi). It's important to note that since KS(j) is derived from its previous KS(m) {m:1..j}, the second round of loop scanning should be calculated from the back to the front. This is because if the derivation is performed from the front to the back, the values saved in the previous loop may be modified, resulting in errors.

```python
import json

class Solution:
    def __init__(self):
        self.vs = [0, 2, 4, 3, 7]
        self.ws = [0, 2, 3, 5, 5]
        self.newResults = [0] * 11

    def test(self):
        result = self.ksp(4, 10)
        print(result)

    def ksp(self, i, c):
        # Start filling in the table
        for m in range(len(self.vs)):
            w = self.ws[m]
            v = self.vs[m]
            for n in range(c, w - 1, -1):
                self.newResults[n] = max(self.newResults[n], self.newResults[n - w] + v)
            # You can output intermediate results here
            print(json.dumps(self.newResults))
        return self.newResults[-1]
```

In this way, we have successfully optimized the space complexity from O(n*c) to O(c). Of course, the cost of space optimization is that we can only know the final result but cannot backtrack the intermediate choices, meaning we cannot find the combination of items we need to select based on the final result.

d. Exact Filling vs. Max Value

The 0-1 knapsack problem generally has two different types of questions: one is the optimal solution of "filling the knapsack exactly," which requires that the knapsack must be filled. In this case, during initialization, except for KS(0) being 0, all other KS(j) should be set to negative infinity. This ensures that the final KS(c) is the optimal solution for filling the knapsack exactly. The other type of question does not require filling the knapsack but only hopes to achieve the largest possible value. In this case, during initialization, KS(0...c) should all be set to 0.

Why is this? Because the initialized array actually represents the legal state in the case where no items can be put into the knapsack. If the knapsack is required to be filled exactly, then only the knapsack with a capacity of 0 can be "filled exactly" without putting anything

in it and having a value of 0. All other capacity knapsacks do not have legal solutions, so they are in an undefined state and should be set to negative infinity. If the knapsack does not need to be filled, then any capacity knapsack has a legal solution, which is "put nothing in it." The value of this solution is 0, so the initial state values are all 0.

4. Complete, Multiple, and 0/1

The Complete Package problem and Multiple Package problem are both variations of the classical 0/1 Knapsack problem.

1. Complete Package (Unbounded) problem:

In the Complete Package problem, also known as the Unbounded Knapsack problem, you are allowed to pick an unlimited number of instances of each item. This means that, unlike the 0/1 Knapsack problem, where you can only choose each item once, you can now pick each item as many times as you want, as long as the capacity of the knapsack is not exceeded.

2. Multiple Package problem:

The Multiple Package problem is a further extension of the Complete Package problem, where each item has a limited number of available instances. In other words, each item has a certain quantity associated with it, and you can only choose up to that quantity of the item. It is a middle ground between the 0/1 Knapsack problem and the Complete Package problem.

3. Comparison:

0/1 Knapsack problem:

- → Each item can be chosen at most once
- → The focus is on maximizing the value of the items chosen, while staying within the knapsack capacity

Complete Package (Unbounded) problem:

- → Each item can be chosen an unlimited number of times
- → The problem is similar to the 0/1 Knapsack problem, but the additional freedom allows for more complex solutions
- → It is generally solved using dynamic programming, with slight modifications to the 0/1 Knapsack problem algorithm

Multiple Package problem:

- → Each item has a limited number of instances available
- → Combines elements of both the 0/1 Knapsack problem and the Complete Package problem, resulting in more complex scenarios
- → Can also be solved using dynamic programming, with further modifications to the algorithm to accommodate the limited quantity of each item

In summary, the Complete Package and Multiple Package problems are extensions of the

classical 0/1 Knapsack problem, with added complexity due to the availability of multiple instances of each item. While the core concepts remain the same, the algorithms for solving these problems need to be adapted accordingly to handle the additional constraints.

5. Case Studies

The following examples and all the code are in the Example Jupyter notebook (https://colab.research.google.com/drive/1UTb6MepHO1DY4auBpBSOHjBAjURDHvhZ?usp=sharing):

LC416.Partition Equal Subset Sum

The problem gives a non-empty array containing only positive integers. The question is whether this array can be divided into two subsets so that the sum of the elements in both subsets is equal.

Since the sum of all elements is known, both subsets should have a sum of sum/2 (so the prerequisite is that sum cannot be an odd number), meaning that the problem can be transformed into selecting some elements from this array so that the sum of these elements is sum/2. If we consider the value of each element as the weight of an item, and the value of each item is 1, then this is an exactly-filled 0/1 knapsack problem.

We define the space-optimized state array dp. Since it is exactly filled, dp[0] should be initialized to 0, and all others should be initialized to INT_MIN. Since the final result is whether the division can be done or not, we can define each element of dp as a boolean type, initialize dp[0] as true, and initialize others as false. The transition equation should use the OR operation instead of the max operation. The complete code is as right shows:

```python
from typing import List

def can_partition(nums: List[int]) -> bool:
    sum_nums = sum(nums)
    n = len(nums)

    if sum_nums % 2 != 0:
        return False

    capacity = sum_nums // 2
    dp = [False] * (capacity + 1)
    dp[0] = True

    for i in range(1, n + 1):
        for j in range(capacity, nums[i - 1] - 1, -1):
            dp[j] = dp[j] or dp[j - nums[i - 1]]

    return dp[capacity]

# Example usage
nums = [1, 5, 11, 5]
print(can_partition(nums))  # Output: True

nums = [1, 2, 3, 5]
print(can_partition(nums))  # Output: False
```

```
True
False
```

LC322.Coin Change

The problem gives an amount value and some denominations, assuming that the number of

coins of each denomination is unlimited. The question is how few coins we can use to make up the given value.

If we consider the denomination as an item, the denomination amount as the weight of the item, and the value of each item as 1, then this problem is an exactly-filled complete package problem. However, this is not about finding the maximum number of items that can be packed but the minimum. We only need to change the max in the previous equation to min, and since it is exactly filled, all other values should be initialized to INT_MAX except dp[0]. The complete code is as follows:

```
[2] from typing import List
    import sys

    def coin_change(coins: List[int], amount: int) -> int:
        dp = [sys.maxsize] * (amount + 1)
        dp[0] = 0

        for i in range(1, len(coins) + 1):
            for j in range(coins[i - 1], amount + 1):
                if dp[j] - 1 > dp[j - coins[i - 1]]:
                    dp[j] = 1 + dp[j - coins[i - 1]]

        return -1 if dp[amount] == sys.maxsize else dp[amount]

    # Example usage
    coins = [1, 2, 5]
    amount = 11
    print(coin_change(coins, amount))  # Output: 3

    coins = [2]
    amount = 3
    print(coin_change(coins, amount))  # Output: -1
```

```
3
-1
```

LC494. Target Sum

This problem gives us an array of non-negative elements and a target value. We need to find the number of situations where adding a positive or negative sign before each number in the array results in an expression equal to the target value S.

Suppose the sum of all elements is sum, the sum of all elements with a positive sign is A, and the sum of all elements with a negative sign is B. Then sum = A + B, and S = A - B. Solving the equation, we get A = (sum + S) / 2. The problem is transformed into selecting some elements from the array so that their sum is exactly (sum + S) / 2. This is an exactly-filled 0/1 knapsack problem, where we need to find the total number of plans. We can change the max in the previously mentioned equation to the sum. Note that the initial value of dp should not be infinity, because we are not seeking the total value but the number of plans, so it should all be initialized to 0 (except for dp[0] initialized to 1). The code is as follows:

```python
from typing import List

def find_target_sum_ways(nums: List[int], S: int) -> int:
    total_sum = sum(nums)
    if S > total_sum or total_sum < -S:
        return 0
    if (S + total_sum) & 1:
        return 0

    target = (S + total_sum) >> 1

    dp = [0] * (target + 1)
    dp[0] = 1
    for i in range(1, len(nums) + 1):
        for j in range(target, nums[i - 1] - 1, -1):
            dp[j] = dp[j] + dp[j - nums[i - 1]]

    return dp[target]

# Example usage
nums = [1, 1, 1, 1, 1]
target_sum = 3
print(find_target_sum_ways(nums, target_sum))  # Output: 5
```

```
5
```

LC474.Ones and Zeroes

The problem gives an array containing only the characters 0 and 1. The task is to select as many strings as possible from the array so that the number of 0s and 1s in these strings does not exceed m and n, respectively.

We can consider each string as an item and the number of 0s and 1s in the string as two types of "weights". So this becomes a two-dimensional 0/1 knapsack problem, with the two weight limits being m and n, aiming to maximize the number of items that can be packed.

We can pre-calculate the two "weights" W0 and W1 for each string and store them in arrays. However, since we only need to use these two values once, we can calculate W0 and W1 when needed instead of using additional arrays to store them. The complete code is as right shows:

```python
from typing import List

def find_max_form(strs: List[str], m: int, n: int) -> int:
    num = len(strs)

    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, num + 1):
        w0, w1 = 0, 0
        # Calculate the two weights of the i-1th string
        for c in strs[i - 1]:
            if c == '0':
                w0 += 1
            else:
                w1 += 1

        # 0/1 Knapsack, reverse iteration to update dp
        for j in range(m, w0 - 1, -1):
            for k in range(n, w1 - 1, -1):
                dp[j][k] = max(dp[j][k], 1 + dp[j - w0][k - w1])

    return dp[m][n]

# Example usage
strs = ["10", "0001", "111001", "1", "0"]
m, n = 5, 3
print(find_max_form(strs, m, n))  # Output: 4
```

```
4
```

6. Conclusion

In conclusion, this paper has provided a comprehensive overview of the 0/1 Knapsack problem, along with its variations, the Complete Knapsack problem and the Multiple Knapsack problem. We discussed their similarities and differences, which mainly lie in the constraints and objectives of each problem. By understanding the fundamental concepts of these problems, we were able to explore various methods for solving them, such as dynamic programming and recursive solutions.

Furthermore, we discussed several optimization techniques, including space and time complexity optimization. These optimizations are crucial for dealing with large datasets and improving the overall performance of the algorithms. We also provided detailed explanations and code implementations for specific example problems, illustrating how to apply these concepts to real-world scenarios.

The examples provided demonstrate the versatility and adaptability of the knapsack problems and their solutions, showcasing how they can be applied to a wide range of situations. The methodologies and concepts presented in this paper serve as a valuable foundation for further exploration and development in the field of combinatorial optimization and related areas.

Overall, the study of the 0/1 Knapsack problem and its variations offers valuable insights into the world of optimization problems, enabling us to better understand and tackle complex real-world challenges.

Citation

https://www.smithsonianmag.com/science-nature/why-knapsack-problem-all-around-us-180974333/
https://www.cnblogs.com/mfrank/p/10587463.html
https://www.cnblogs.com/mfrank/p/10533701.html
https://www.cnblogs.com/mfrank/p/10849505.html
https://zhuanlan.zhihu.com/p/93857890
https://leetcode.com/discuss/study-guide/1200320/Thief-with-a-knapsack-a-series-of-crimes
https://leetcode.com/problems/last-stone-weight-ii/solutions/294888/JavaC++Python-Easy-Knapsacks-DP/