discount tags hackerrank solution

Understanding the Discount Tags Problem on HackerRank

The "Discount Tags" problem on HackerRank is a classic challenge designed to test your understanding of dynamic programming and efficient algorithmic thinking. Many aspiring competitive programmers encounter this problem as they hone their skills, seeking effective solutions and explanations. This article delves deep into the discount tags hackerrank solution, providing a comprehensive guide that covers the problem's nuances, common pitfalls, and optimal approaches. Whether you're looking for a detailed breakdown of the logic, efficient code implementations, or strategies to tackle similar dynamic programming challenges, you'll find valuable insights here. We will explore various techniques to arrive at the most efficient discount tags hackerrank solution, ensuring you grasp the underlying principles and can apply them to future problems.

Table of Contents

Introduction to the Discount Tags Problem

The Discount Tags problem presents a scenario where you need to calculate the minimum cost to buy a set of items, given that certain combinations of items can be purchased together at a discounted price. This isn't a simple greedy problem; the interdependencies between discounts and item choices make dynamic programming the most suitable approach. The goal is to strategically select which discounts to apply to minimize the overall expenditure. Understanding how to model this problem as a state transition is crucial for devising an effective discount tags hackerrank solution. Many beginners struggle with defining the state and the transitions, often leading to TLE (Time Limit Exceeded) errors or incorrect results. This guide aims to demystify these complexities and provide a clear path towards a correct and efficient discount tags hackerrank solution.

Deconstructing the Discount Tags Problem

At its heart, the Discount Tags problem involves making a series of decisions that impact future choices. You have a list of items, each with a base price. Additionally, you have a set of discount "tags" that can be applied. Each tag applies to a specific subset of items and reduces their combined cost. The challenge lies in selecting which tags to use, considering that a tag can only be used once, and applying a tag might make another tag redundant or suboptimal. The problem statement typically defines the items, their prices, and the available discount tags, specifying the items covered by each tag and the discount amount or final price. A robust discount tags hackerrank solution must account for all these parameters and their interactions.

Understanding the Constraints and Input Format

Before diving into solutions, it's vital to understand the typical constraints and input format for the Discount Tags problem on HackerRank. This usually involves the number of items, the number of discount tags, the base prices of each item, and for each tag, the set of items it applies to and the discounted price. Constraints on these numbers will dictate the feasibility of certain algorithmic approaches. For example, if the number of items is small, brute force might be considered, but for larger inputs, efficiency becomes paramount. Understanding these limitations is the first step towards designing an appropriate discount tags hackerrank solution.

Identifying Overlapping Subproblems

The hallmark of dynamic programming problems is the presence of overlapping subproblems. In the Discount Tags problem, consider the decision of whether to apply a particular discount tag. This decision affects which items are still available for other discounts and influences the remaining cost. The optimal way to purchase a subset of items, given a set of available discount tags, is a subproblem that will be encountered multiple times in different contexts. Recognizing these recurring subproblems is key to building a dynamic programming solution for discount tags. The efficiency of a discount tags hackerrank solution often hinges on how well it exploits these overlapping subproblems.

Key Concepts for the Discount Tags Solution

Several core computer science concepts are fundamental to solving the Discount Tags problem effectively. These include understanding bit manipulation for representing subsets of items, dynamic programming principles, and state-space representation. A good discount tags hackerrank solution will leverage these concepts to manage complexity and achieve optimal performance.

Bitmasking for Item Subsets

Given that the number of items is often limited (e.g., up to 20 items), bitmasking is a highly effective technique to represent subsets of items. Each bit in an integer can represent the inclusion or exclusion of an item. For instance, if you have 5 items, a 5-bit integer can represent any combination of these items. A bit set to 1 means the item is included in the subset, while a bit set to 0 means it's not. This allows for a compact and efficient way to manage the state of which items have been considered or purchased, which is crucial for a scalable discount tags hackerrank solution.

Dynamic Programming State Definition

Defining the correct state for dynamic programming is arguably the most critical step. A typical state for the Discount Tags problem might involve `dp[mask]`, representing the minimum cost to purchase the set of items represented by the `mask`. The `mask` would be a bitmask where the i-th bit is set if the i-th item has been purchased. Alternatively, the state could represent the minimum cost to purchase a subset of items, and the transitions would involve either buying an item individually or using a discount tag. The choice of state definition directly impacts the complexity and elegance of the discount tags hackerrank solution.

Transitions and State Updates

Once the state is defined, the next step is to establish the transitions. How do you move from one state to another? For `dp[mask]` representing the cost for items in `mask`, a transition could involve considering a new item `i` not yet in `mask`. The new state would be `mask | (1 <

Developing the Discount Tags HackerRank Solution

Crafting an efficient and correct discount tags hackerrank solution involves a systematic approach, starting from understanding the problem thoroughly and moving towards implementation details. The core lies in dynamic programming, but proper state management and transition logic are key.

Iterative vs. Recursive DP

Dynamic programming solutions can often be implemented iteratively or recursively with memoization. For the Discount Tags problem, an iterative approach is generally preferred due to potential stack overflow issues with deep recursion and often better performance. An iterative DP solution would typically build up the `dp` table from smaller subsets of items to larger ones. This means `dp[mask]` would be computed based on previously computed values for masks with fewer items set.

Base Cases and Initialization

Every dynamic programming solution requires correctly defined base cases. For the Discount Tags problem, `dp[0]` (representing the cost to purchase no items) is usually initialized to 0. All other `dp` states should be initialized to infinity (a very large number) to signify that these states are initially unreachable or have an infinitely high cost. This ensures that the first valid calculation for a state correctly sets its minimum cost.

Memoization for Overlapping Subproblems

If opting for a recursive approach, memoization is crucial. A memoization table (often an array or map) stores the results of expensive function calls and returns the cached result when the same inputs occur again. For the Discount Tags problem, this table would likely be indexed by the bitmask representing the subset of items already purchased. This prevents redundant computations, making the solution efficient.

Common Approaches to the Discount Tags Problem

While dynamic programming is the primary method, understanding how it's applied in practice for discount tags is essential. Different DP formulations can exist, each with its own trade-offs in terms of complexity and ease of implementation.

Top-Down DP (Memoization)

A top-down approach uses recursion with memoization. The function `solve(mask)` would return the minimum cost to purchase items represented by `mask`. Inside `solve(mask)`, we first check if the result for `mask` is already computed. If so, return it. Otherwise, iterate through all possible ways to form `mask`: either by adding a single item to a smaller subset, or by applying a discount tag that covers a subset of items within `mask`. The minimum cost is then stored and returned.

Bottom-Up DP (Tabulation)

A bottom-up approach builds the DP table iteratively. Starting with `dp[0] = 0` and all other `dp` values as infinity, we iterate through all possible masks from 0 to `(1 <

- Adding a single item: For each item `i` not in `mask`, we can transition to `mask | (1 <
- Applying a discount tag: If a discount tag applies to a set of items `discount_mask`, and `(mask & discount_mask) == mask` (meaning all items in the discount are already covered by `mask`), we can potentially achieve `mask | discount_mask` with a reduced cost. This formulation often needs careful handling of how discounts are applied. A more direct bottom-up approach might iterate through masks and consider which discount tags can be used to achieve that mask from a smaller set of already purchased items.

This iterative process ensures that when we compute `dp[mask]`, all necessary subproblem solutions are already available. This is a very common and effective structure for a discount tags hackerrank solution.

Dynamic Programming: The Core of the Discount Tags Solution

Dynamic programming is the cornerstone of solving the Discount Tags problem efficiently. Its ability to break down a complex problem into smaller, overlapping subproblems and store their solutions makes it ideal for scenarios with combinatorial choices like this.

State Space and Transitions Explained

Let `n` be the number of items. The state space can be represented by `2^n` possible subsets of items. Each state `dp[mask]` will store the minimum cost to acquire exactly the set of items represented by the bitmask `mask`.
The transitions work as follows:
To calculate `dp[mask]`, we can consider all possible last steps that led to purchasing the items in `mask`.

- **Buying the last item individually:** If item `i` is the last item added to form `mask` (i.e., `(mask >> i) & 1` is true, and `mask ^ (1 <
- **Applying a discount tag:** If a discount tag covers items in `discount_mask`, and `mask` is exactly `discount_mask`, then `dp[mask]` can be `discount_price`. If `mask` is a superset of `discount_mask`, and `prev_mask = mask ^ discount_mask`, then we can potentially update `dp[mask]` using `dp[prev_mask] + discount_price` if all items in `discount_mask` were not part of `prev_mask`. The logic here is that a discount tag is applied to a specific set of items, and the problem statement dictates if it can be applied to items already purchased or if it's a bundle deal for a new set.

A more streamlined DP approach often involves iterating through masks from 0 up to `(1<> i) & 1)`), we can transition to state `mask | (1 <For each discount tag `d` with items `discount_items_mask` and price `discount_price`: If `(mask & discount_items_mask) == 0` (meaning no items in the discount set are already in `mask`), we can transition to `mask | discount_items_mask` with cost `dp[mask] + discount_price`. This latter transition is more common when discounts are for bundles of new items.
The crucial part of a good discount tags hackerrank solution is correctly modeling the application of discounts, which often means they are applied to specific sets of items and not just adding cost savings to existing purchases. The problem statement usually implies that a discount is a way to achieve a set of items at a certain price.

Complexity Analysis of DP Approaches

A typical DP approach using bitmasks will have a state space of `O(2^n)`, where `n` is the number of items. For each state, we might iterate through `n` items or `m` discount tags.

- If we consider adding single items: Each state transition takes `O(n)` time. Total time: `O(n 2^n)`.
- If we consider applying discount tags: If there are `m` discount tags, and each tag covers a subset, checking and applying a tag could take time related to the number of items in the tag. A simpler model where each discount has a fixed `discount_mask` and `discount_price` makes transitions `O(m)`. Total time: `O(m 2^n)`.

The overall time complexity is often dominated by `O(n 2^n)` or `O(m 2^n)`. The space complexity is `O(2^n)` to store the DP table.

Optimizing the Discount Tags HackerRank Solution

Achieving an optimal discount tags hackerrank solution often requires careful attention to implementation details and potentially exploring minor optimizations if the constraints allow.

Preprocessing Discount Tags

Before starting the DP, it can be beneficial to preprocess the discount tags. For example, if multiple discount tags cover the exact same set of items, you only need to consider the one with the lowest price. Also, if a discount tag's price is higher than the sum of individual item prices it covers, it's effectively useless and can be discarded. This preprocessing can sometimes prune the search space or simplify the transitions.

Efficient Bitwise Operations

Since bitmasking is central, ensuring efficient use of bitwise operations is key. Operations like `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<>` (right shift) are generally fast. Understanding how to quickly check if an item is in a mask (`(mask >> i) & 1`), or to count the number of set bits (population count), can be useful. For example, checking if a discount mask `d_mask` is a subset of a current mask `curr_mask` is `(curr_mask & d_mask) == d_mask`. Conversely, checking if `d_mask` is disjoint from `curr_mask` is `(curr_mask & d_mask) == 0`.

Pruning Unnecessary States

In some DP problems, it's possible to prune states that are clearly suboptimal. For example, if you have two ways to reach the same `mask`, and one way involved a significantly higher cost, you might not need to explore further from the higher-cost state if it's guaranteed to lead to worse overall solutions. However, for the Discount Tags problem with a standard DP formulation on masks, this kind of pruning is often implicitly handled by always taking the minimum cost to reach a state.

Illustrative Example of the Discount Tags Solution

Let's consider a small example to illustrate the DP approach for discount tags.

Problem Setup

Items:

- Item 0: Price 10
- Item 1: Price 15
- Item 2: Price 20

Discount Tags:

- Tag A: Items {0, 1}, Price 20
- Tag B: Items {1, 2}, Price 30

We want to find the minimum cost to buy all items {0, 1, 2}.

DP Table Construction

Let `n=3`. Masks will range from 0 (000) to 7 (111).
`dp[mask]` = minimum cost to buy items represented by `mask`.
Initialize `dp[0] = 0`, all others to infinity.

Iteration 1 (mask = 0): `dp[0] = 0`.

- Add item 0: `dp[1] = min(inf, dp[0] + 10) = 10` (001)
- Add item 1: `dp[2] = min(inf, dp[0] + 15) = 15` (010)
- Add item 2: `dp[4] = min(inf, dp[0] + 20) = 20` (100)
- Tag A ({0, 1}, price 20): `dp[3] = min(inf, dp[0] + 20) = 20` (011)
- Tag B ({1, 2}, price 30): `dp[6] = min(inf, dp[0] + 30) = 30` (110)

Iteration 2 (mask = 1, i.e., item 0 purchased): `dp[1] = 10`.

- Add item 1: `dp[3] = min(dp[3], dp[1] + 15) = min(20, 10 + 15) = 20` (already 20, no change)
- Add item 2: `dp[5] = min(inf, dp[1] + 20) = min(inf, 10 + 20) = 30` (101)
- Tag A ({0, 1}, price 20): Cannot apply to mask 1 as item 0 is already there and this formulation considers adding disjoint sets. A better way: If `mask` is a subset of `discount_mask`, and `discount_mask` is the target, then `dp[discount_mask] = min(dp[discount_mask], dp[mask] + discount_price)`. More commonly, `dp[mask | discount_mask] = min(dp[mask | discount_mask], dp[mask] + discount_price)` if items in discount are not already in mask.
- Tag B ({1, 2}, price 30): Cannot apply to mask 1.

Let's refine the transition logic for bottom-up DP for clarity:

Initialize `dp` array of size `1 <Iterate `mask` from 0 to `(1 <
If `dp[mask]` is infinity, continue.

Option 1: Add single items

For `i` from 0 to `n-1`:

If item `i` is not in `mask` (`!((mask >> i) & 1)`):

`next_mask = mask | (1 <
`dp[next_mask] = min(dp[next_mask], dp[mask] + item_price[i])`

Option 2: Apply discount tags

For each discount tag `d` with `d_mask` and `d_price`:

If the items in `d_mask` are disjoint from `mask` (`(mask & d_mask) == 0`):
`next_mask = mask | d_mask`
`dp[next_mask] = min(dp[next_mask], dp[mask] + d_price)`

Following this refined logic:
Initial: `dp = [0, inf, inf, inf, inf, inf, inf, inf]`
mask=0: `dp[0]=0`

- Add 0: `dp[1] = min(inf, 0+10) = 10`
- Add 1: `dp[2] = min(inf, 0+15) = 15`
- Add 2: `dp[4] = min(inf, 0+20) = 20`
- Tag A {0,1}, price 20: `dp[3] = min(inf, 0+20) = 20`
- Tag B {1,2}, price 30: `dp[6] = min(inf, 0+30) = 30`

mask=1 (001, item 0): `dp[1]=10`

- Add 1: `dp[3] = min(20, 10+15) = 20`
- Add 2: `dp[5] = min(inf, 10+20) = 30`
- Tag A {0,1}: Disjoint with {0}? No. If `d_mask` can be applied to items already purchased to get a better price for `mask | d_mask`, it's `dp[mask | d_mask] = min(dp[mask | d_mask], dp[mask] + d_price)`. But this is not quite right for the common formulation. It's usually about achieving the `d_mask` itself or a superset. Let's stick to the interpretation: "for each `mask`, consider how to expand it".
  Tag A ({0,1}, 20): If `mask` is a subset of `d_mask`, and we want to reach `d_mask`, `dp[d_mask] = min(dp[d_mask], dp[mask] + d_price)`. This is still not ideal.

A cleaner bottom-up DP often looks like this: Iterate through masks. For each `mask`, consider all discount tags. If a discount tag `d_mask` can be completed by adding items currently not in `mask` to form `d_mask`, that's one way. Or, if `d_mask` is a subset of `mask`, we can update `dp[mask]` by considering a previous state `mask \ d_mask`.
The most common approach: `dp[mask]` = min cost for exactly the items in `mask`.
Base case `dp[0]=0`.
Iterate `mask` from `0` to `(1<