Greedy Algorithms

Presented by Alexander Mak

What does it mean to be greedy?

- Greedy algorithms are a class of algorithms that make locally optimal choices at each stage with the hope of finding a global optimum.
- Make best possible decision at each step without considering future consequences.
- Key characteristics:
  - **Greedy Property:** Global optimum can be reached by selecting a local optimum at each step.
  - **Optimal Substructure:** The solution to the problem can be constructed from optimal solutions to its subproblems.
  - **No Backtracking:** Once a decision is made, it is never reconsidered or undone.
- Some of these characteristics are shared by dynamic programming problems, which can sometimes make it difficult to differentiate. The difference is the necessity of backtracking to reconsider different options.

Presented by Alexander Mak

Greedy Example: U.S. Coin Change Problem

Imagine you are a cashier, and your goal is to give a customer the minimum number of coins as change for a given amount.

You have an unlimited supply of coins with certain denominations (e.g., quarters, dimes, nickels, and pennies), and you want to find the minimum number of coins for the total amount of change.

Greedy Choice: At each step, you choose the largest possible coin denomination that is less than or equal to the remaining amount of change.

No Backtracking: Once a coin denomination is chosen, you don't reconsider or undo that choice.

Optimal Substructure: The optimal solution for the entire amount of change can be constructed by combining the optimal solutions for the subproblems (smaller amounts of change).

We are are able to be greedy because each coin is a multiple of the next smaller coin, ensuring that the greedy choice at each step leads to the overall minimum number of coins.

Presented by Alexander Mak

Greedy doesn’t always work…

Let’s try the previous example again, except this time we can be given ANY denomination in our list instead of just US coins where larger coins are guaranteed to be able to be made from smaller denominations.

The greedy algorithm, as described earlier with specific coin denominations that are multiples of each other, doesn't always work optimally for the general case. This general case is solved using dynamic programming.

Consider denominations [1, 3, 4] and the target amount is 6. The optimal solution is two coins of 3, but a greedy algorithm that always chooses the largest coin value might select one coin of 4 and two coins of 1, which is not optimal.

Comparing Greedy and Dynamic Programming

	Specific Coin Denominations (Greedy Approach):	**Arbitrary Coin Denominations (DP Approach):**
Structure:	Fixed set of denominations, and each coin's value is a multiple of the next smaller coin.	Coin denominations can be arbitrary, without the guarantee that each coin's value is a multiple of the next smaller coin.
Backtracking needed?	No	Yes
Approach:	A greedy algorithm can be used, making locally optimal choices at each step without the need for backtracking.	DP approach is often necessary. Backtracking might be required. If the greedy algorithm chooses a larger coin early on, it might miss a combination of smaller coins that result in a minimum number of coins for a certain amount.

Example code

```
def greedy_coin_change(amount):
    coins = [25, 10, 5, 1]  # Quarters, Dimes, Nickels, Pennies
    total_coins = 0
    change = []

    for coin in coins:
        num_of_coins = amount // coin
        total_coins += num_of_coins
        amount -= num_of_coins * coin
        change.append((coin, num_of_coins))

    return total_coins, change
```

Questions?

Let’s practice!

https://github.com/Dijkstra-LLC/dsa_live_pro/tree/main/W03D02/classwork