Data Structure and Algorithm 2

Coin Change Problems(Dynamic Programming)

Tahmid Mosaddeque

UIU

August 23, 2025

So far, we have covered-

- Dynamic Programming Properties:
  - Overlapping Subproblems
  - Optimal Substructure
- 0/1 Knapsack problem formulation
- Top Down solution
- Bottom Up solution

Problem 1

Statement

You are given an array of k coin denominations. Your task is to determine whether you can pay a bill of n dollars using these denominations. You are allowed to use coins from a single denomination multiple times.

For example, consider the denominations 3\$, 6\$, 9\$ and a bill of 19\$. In this case, you cannot pay 19\$ using the given denominations. However, you can pay 12\$ by using 9\$ and 3\$.

Solution

- We can pay n\$ only if we can pay any of the smaller bills of $(n - coins[1])$\$ or $(n - coins[2])$\$ or ... $(n - coins[k])$\$ (Optimal Substructure Property)
- If none of the smaller bills are payable, then we cannot make n\$ using the given denominations.
- The smallest subproblem involves 0 dollars, which is always payable. We do not need to take any coins.
- Recursive solution:
  $f(n) = f(n - coins[1])\ or\ f(n - coins[2])\ or \ldots f(n - coins[k])$

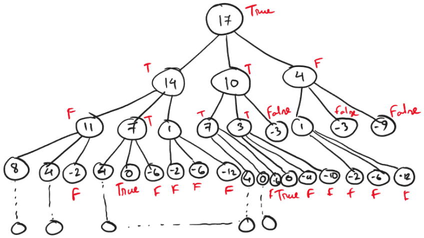
Recursive implementation

```
def isPossible(n):
    if n < 0:
        return False
    #There is no way to make a negative amount
    if n == 0:
        return True
    flag = False
    for i in range(len(coins)):
        flag = flag or isPossible(n-coins[i])
    return flag
```

Example

Suppose you have 3 denominations of bills, [3$, 7$, 13$]. Determine if it is possible to pay 17$

Recursion Tree(Without DP):



Therefore, it is possible to pay 17$

Time Complexity

- The height of the recursion tree depends on the values of the denominations. In the worst-case scenario, we can assume there is a 1 denomination. In this case, the height of the recursion tree will be n.
- The branching factor is k, as we have k different denominations.
- Therefore, the worst-case time complexity will be $O(k^n)$.

DP implementation

```
memo = [-1]*n
#-1 is used as a flag
to indicate that no value has been calculated
def isPossible(n):
    if n < 0:
        return False
    if n == 0:
        return True
    #Memoization
    if memo[n] != -1:
        return memo[n]
    flag = False
    for i in range(len(coins)):
        flag = flag or isPossible(n-coins[i])
    #Save the value for the future.
    memo[n] = flag
    return flag
```

Time Complexity With DP

- There are n possible states and each state is calculated exactly once because of memoization.
- A loop for k denominations is used to find the value of each state.
- Therefore, the total time complexity is $O(nk)$

Problem 2

Statement

Given a set of coin denominations and a target amount, find the minimum number of coins needed to make that amount.

If you have coins of 1, 5, and 10 cents, and you want to make 12 cents. There are several possibilities:
$12 = 1 + 1 + ... + 1$ (12 coins)
$12 = 1 + 1 + 5 + 5$ (4 coins)
$12 = 1 + 1 + 10$ (3 coins) Therefore, the optimal solution in this case is to use 3 coins.

Solution

- To find the minimum number of coins required to make n. First, we need to calculate the minimum number of coins required to make $(n - coins[1])$, $(n - coins[2])$, ... $(n - coins[k])$
- Let's assume, among all the subproblems $(n - coins[i])$ requires the minimum number of coin. So, we can add another $coins[i]$ to make n using the minimum number of coins.
- Recursive solution:
  $f(n) = 1 + min\{f(n - coins[1]), f(n - coins[2]), ..., f(n - coins[k])\}$

Recursive implementation

```
def minimumCoins(n):
    if n < 0:
        return infinity
    # Impossible to make a negative amount
    if n == 0:
        return 0
    # We don’t need any coins to make 0
    minimum = infinity
    for i in range(len(coins)):
        minimum = min(minimum, minimumCoins(n-coins[i]))
    return 1 + minimum
```

Example

Suppose you have an infinite number of 2c, 3c and 7c.

1. Draw the recursion tree to determine the minimum number of coins required to make 15c.
2. Write a pseudocode for the above problem using dynamic programming(Memoization).
3. Analyze the time complexity of your DP algorithm.

Other variations(Try yourself)

Given *coins* $= [2, 3, 5]$, how many ways can we make 10?

Problem 3

In this variation, the combinations $10 = 2 + 3 + 5$ and $10 = 5 + 3 + 2$ will be considered as separate ways to represent the number 10.

Problem 4

In this variation, the combinations $10 = 2 + 3 + 5$ and $10 = 5 + 3 + 2$ will be considered as the same way to represent 10.