Dynammic Programming 60

Saturday, 11 November 2023 5:42 PM

① count number of coins required to make a given value.
(assume infinite supply of coins)

Recurrence Relation

count(coins, n, sum) = count(coins, n, sum − coins[n−1]) + count(coins, n−1, sum)

For each coin there are two options:
① Include the current coin
② Exclude the current coin

Base case
① If target sum is 0, only one way, by not choosing any coins : return 1
② If target sum is negative or no more coins to consider, then there are no ways to make the sum : return 0

Time : $O(2^{sum})$
Space : $O(sum)$

```
def count(self, coins, n, sum):
    If sum == 0:
        return 1
    If sum < 0 :
        return 0
    If n <= 0:
        return 0
    return self.count(coins, n, sum-coins[n-1])
           +
           self.count(coins, n-1, sum)
```

Memoization

above recursive solution has optimal substructure and overlapping subproblems
so DP can be used.
2D array can be used to store results of previously solved subproblems

dp[i][j] represents the no. of distinct ways to make the sum j by using first i coins.

Time : $O(N \times sum)$
Space : $O(N \times sum)$

```
def count(self, coins, N, sum):
    def countDP(coins, n, sum, dp):
        If sum == 0:
            dp[n][sum] = 1
            return dp[n][sum]
        If n <= 0 or sum < 0:
            return 0
        If dp[n][sum] != -1:
            return dp[n][sum]
        dp[n][sum] = countDP(coins, n, sum-coins[n-1], dp)
                     +
                     countDP(coins, n-1, sum , dp)
        return dp[n][sum]
    dp = [[-1 for i in range(sum+1)] for j in range(N+1)]
    ans = countDP(coins, N, sum, dp)
    return ans
def count(self, coins, N, sum):
    dp = [[0 for i in range(sum+1)] for j in range(N+1)]
    dp[0][0] = 1
    for i in range(1,N+1):
        for j in range(sum+1):
            dp[i][j] += dp[i-1][j]
            if j - coins[i-1] >= 0:
                dp[i][j] += dp[i][j-coins[i-1]]
    return dp[N][sum]
```

Tabulation

— 2D array with rows and columns equal to no. of coin denominations and target sum.
— dp[N][N] will be ans as it will represent base case, where target sum is 0.
— Iterate through array from i = 1 to N, representing current coin being considered:
  — Inner loop iterates over target sum (j from 0 to sum)
    — add no. of ways to make change without using current coin.
      dp[i][j] += dp[i-1][j]
    — add no. of ways to make change using current coin.
      dp[i][j] += dp[i][j-coins[i-1]]
— dp[N][sum] will contain total no. of ways to make change for given target sum using available coin denominations.

Space Optimized
— dp[i] represents no. of ways to make sum i using given coin denominations.
— outer loop iterates over coins, and inner loop iterates over target sum.
— For each dp[j], calculate no. of ways to make change using current coin denominations and previous results stored in dp.
— dp[sum] contains total no. of ways to make change for the given target sum using available coins.

```
def count(self, coins, N, sum):
    dp = [0 for i in range(sum+1)]
    dp[0] = 1
    for i in range(N):
        for j in range(coins[i], sum+1):
            dp[j] += dp[j-coins[i]]
    return dp[sum]
```

② Find min no. of coins to make a given value.
— Instead of finding total no. of possible combinations, we need to find combination with min no. of coins.
  If value >= 0 ; 0 coins required
  If value > 0 : minCoins(coins, value) = min(1 + minCoins(value−coins[i]))

Recursion

```
def minCoins(coins, M, amount):
    dp = [float('inf')] * (amount+1)
    dp[0] = 0
    for a in range(1, amount+1):
        for coin in coins :
            If a - coins >= 0:
                dp[a] = min(dp[a], 1+dp[a-coins])
    return dp[amount] If dp[amount] != float('inf') else -1
```

③ 0/1 Knapsack Problem

Recursion

Simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all subsets, pick the subset with max profit.

Optimal Substructure :
Case 1 : Item is included in optimal subset.
Case 2 : Item is not included in optimal subset.
Hence, maximum profit can be the max of following two values:
Case 1 : (Include n^{th} item) Value of nth item plus max value obtained by remaining n-1 items and weight-(w_n)-weight of nth item
Case 2 : (Exclude n^{th} item) Max value obtained by n-1 items and W weight.
— If weight of N item is greater than W, then the nth item can not be included and case 2 is only possible.

```
def knapSack(profit, wt, w, n):
    If w == 0 or n == 0:
        return 0
    If wt[n-1] >= 0:
        return self.knapSack(profit, wt, w, n-1)
    return max(profit[n-1] + knapSack(profit, wt, w-wt[n-1], n-1),
               knapSack(profit, wt, w, n-1))
```

Time : $O(2^N)$
Space : $O(N)$ — stack space required

DP / Memoization Approach
— Compute same subproblems
Note → If we get a subproblem the first time, we can store this problem by creating a 2D array that can store a particular state (n,w).
If we come across same state (n,w) again instead of calculating it in exponential complexity, we simply return the result stored in table in constant time.

```
def knapSack(self, wt, wt, wt, w, n):
    If w == 0 or n == 0: return 0
    dp = [[-1 for i in range(w+1)] for j in range(n+1)]
    If dp[n][w] != -1: return dp[n][w]
    If wt[n-1] <= w:
        dp[n][w] = max(knapSack(wt, wt, w-wt[n-1], n-1),
                       knapSack(wt, wt, w, n-1))
    return dp[n][w]
```

```
def knapSack(profit, wt, w, n):
    dp = [[-1 for i in range(w+1)] for j in range(n+1)]
    for i in range(1,n+1):
        for w in range(w+1):
            If i == 0 or w == 0:
                dp[i][w] = 0
            Elif wt[i-1] <= w:
                dp[i][w] = max(profit[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w])
            else:
                dp[i][w] = dp[i-1][w]
    return dp[n][w]
```

Time : $O(N \times W)$
Space : $O(N \times W)$

Bottom Up
— re-computation of the same problem can be avoided by constructing a temporary array K[][] in a bottom-up manner.

Space Optimized
— For calculating current rows of dp[] array, we require only previous row, but we start traversing the row from right to left then it can be done with a single row.

```
def knapSack(profit, wt, w, n):
    dp = [0]*w [0]*(w+1)
    for i in range(n):
        for j in range(w, wt[i]-1):
            If wt[i] <= j:
                dp[j] = max(dp[j], val[i]+dp[j-wt[i]])
    return dp[w]
```

weights[] = {1, 3, 4} W = 6
profits[] = {10, 15, 20}

	0	1	2	3	4	5	6
	0	0	0	0	0	0	0
{1} 1	0	10	10	10	10	10	10
{1,3} 3	0	10	10	10	10	15	25
{1,3,4} 4	0						

weights
max(10, 15) = 15
max(10, 10+15) = 25

0	1	2	3	4	5	6

dp[3] = max(dp[3], dp[n−wt[i−1]] + val[i−1])

④ Binomial Coefficient C(n,k)
— defined as coefficient of $x^n k$ in the expansion of $(1+x)^n$
— Binomial coefficient C(n,k) also gives no. of ways, disregarding order, that k objects can be chosen from among n objects without replacement, i.e. no. of k-elements subsets (or k-combinations) of a n-element set.

1) Optimal Substructure

C(n,k) = C(n−1, k−1) + C(n−1, k)
C(n,0) = C(n,n) = 1

```
def n(n,k):
    If k > n :
        return 0
    If k == 0 or k == n:
        return 1
    return n(n-1,k-1)+n(n-1,k)
```

⑤ Permutation Coefficient
P(n,k) → no. of ways to choose an ordered subset having k elements from a set of n elements

$$P(n,k) = n \times (n-1) \times \cdots \times (n-k+1) = \begin{cases} 0, & \text{if } k > n \\ \frac{n!}{(n-k)!}, & \text{otherwise} \end{cases}$$

P(n,k) = P(n−1, k) + k * P(n−1, k−1)

O(n) time / O(1) space

```
def n!k(n,k):
    f = 1
    for i in range(k):
        f = f * (n-i)
    return f
```

⑥ Program for n^{th} Catalan number
— defined as mathematical sequence that consists of positive integers, which can be used to find the number of possibilities of various combinations.
— denoted by $C_n = \frac{(2n)!}{(n+1)!\,n!}$

Recursive formula → $C_0 = 1$; $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$ for $n \geq 0$