13) (Coin change problem) Given a set of coins and amount, Write a dynamic programming algorithm to find out how many ways we can make the change of the amount using the coins given. Draw the table using Amount = 5 coins [] = {1,2,3}. Specify time and space complexity.

CoinChangeWays(coins, amount):

# Initialize the dp array

dp = array of size (amount + 1) with all values initialized to 0

dp[0] = 1 # Base case: one way to make 0 amount

# Build the dp table

for coin in coins:

for i from coin to amount:

dp[i] += dp[i - coin]

return dp[amount]

Zaman Karmaşıklığı: O(n \* amount), burada n farklı paraların sayısıdır ve miktar(amount) hedef miktardır. Bunun nedeni, her bir alt miktar için her bir parayı işliyor olmamızdır.

Space Complexity: O(amount), çünkü alt problemlerin sonuçlarını depolamak için amount + 1 boyutunda tek bir dizi kullanıyoruz.

14)

(Çubuk kesme problemi) Diyelim ki n uzunluğunda bir çubuğunuz var ve çubuğu kesmek istiyorsunuz.

ve parçaları, alacağınız toplam para miktarını maksimuma çıkaracak şekilde satabilirsiniz. i uzunluğundaki bir parça pi dolar değerindedir. Uzunluğu n olan bir çubuğu kesmenin kaç yolu vardır? (Bruteforce karmaşıklığı?) Yaz Aldığınız toplam para miktarını maksimuma çıkaran dinamik bir programlama algoritması yazın. {1,2,3,4} uzunluğu için sırasıyla n=5 ve prices={2,5,7,8} değerlerini kullanarak tabloyu çizin.

RodCutting(prices, n):

# Initialize the dp array

dp = array of size (n + 1) with all values initialized to 0

# Fill the dp table

for i from 1 to n:

max\_val = -1

for j from 1 to i:

max\_val = max(max\_val, prices[j-1] + dp[i - j])

dp[i] = max\_val

return dp[n]

**Example with n = 5 and prices = {2, 5, 7, 8}**

Here, the prices array represents the prices for lengths 1, 2, 3, and 4 respectively.

**Initialization:**

* **dp = [0, 0, 0, 0, 0, 0]**

**Filling the DP Table:**

1. **For i = 1:**

dp[1] = max(dp[1], prices[0] + dp[0]) = max(0, 2 + 0) = 2

dp = [0, 2, 0, 0, 0, 0]

1. for i=2

dp[2] = max(dp[2], prices[0] + dp[1]) = max(0, 2 + 2) = 4

dp[2] = max(dp[2], prices[1] + dp[0]) = max(4, 5 + 0) = 5

dp = [0, 2, 5, 0, 0, 0]

3.for i=3

dp[3] = max(dp[3], prices[0] + dp[2]) = max(0, 2 + 5) = 7

dp[3] = max(dp[3], prices[1] + dp[1]) = max(7, 5 + 2) = 7

dp[3] = max(dp[3], prices[2] + dp[0]) = max(7, 7 + 0) = 7

dp = [0, 2, 5, 7, 0, 0]

4.for i=4

dp[4] = max(dp[4], prices[0] + dp[3]) = max(0, 2 + 7) = 9

dp[4] = max(dp[4], prices[1] + dp[2]) = max(9, 5 + 5) = 10

dp[4] = max(dp[4], prices[2] + dp[1]) = max(10, 7 + 2) = 10

dp[4] = max(dp[4], prices[3] + dp[0]) = max(10, 8 + 0) = 10

dp = [0, 2, 5, 7, 10, 0]

5.for i=5

dp[5] = max(dp[5], prices[0] + dp[4]) = max(0, 2 + 10) = 12

dp[5] = max(dp[5], prices[1] + dp[3]) = max(12, 5 + 7) = 12

dp[5] = max(dp[5], prices[2] + dp[2]) = max(12, 7 + 5) = 12

dp[5] = max(dp[5], prices[3] + dp[1]) = max(12, 8 + 2) = 12

dp = [0, 2, 5, 7, 10, 12]

15) (Edist distance problem) The minimum edit distance between two strings is defined as the minimum number of editing operations (insertion, deletion, substitution) needed to transform one string into another. Write a dynamic programming algorithm to find minimum edit distance between two strings. (The cost of deletion and insertion is +1, substitution is +2, match is 0.) Draw the table using “intention” and “execution” strings.

FUNCTION min\_edit\_distance(s1, s2):

m = LENGTH(s1)

n = LENGTH(s2)

// Initialize the 2D array (dp) with dimensions (m+1) x (n+1)

dp = ARRAY(m + 1, n + 1)

// Base cases: transforming an empty string

FOR i FROM 0 TO m:

dp[i][0] = i

FOR j FROM 0 TO n:

dp[0][j] = j

// Fill the dp table using the recurrence relation

FOR i FROM 1 TO m:

FOR j FROM 1 TO n:

IF s1[i - 1] == s2[j - 1]:

dp[i][j] = dp[i - 1][j - 1] // Characters match, no cost

ELSE:

dp[i][j] = MIN(dp[i - 1][j] + 1, // Deletion

dp[i][j - 1] + 1, // Insertion

dp[i - 1][j - 1] + 2) // Substitution

// The value in dp[m][n] is the minimum edit distance

RETURN dp[m][n]

---

e x e c u t i o n

0 1 2 3 4 5 6 7 8 9

i 1 2 3 4 5 6 7 7 8 9

n 2 3 4 5 6 7 7 8 9 9

t 3 4 5 6 6 6 7 8 9 10

e 4 3 4 5 6 7 8 9 10 11

n 5 4 5 6 7 8 9 10 11 12

t 6 5 6 7 8 9 9 10 11 12

i 7 6 7 8 9 10 11 10 11 12

o 8 7 8 9 10 11 12 11 10 11

n 9 8 9 10 11 12 13 12 11 10Topluluk Tarafından Doğrulandı simgesi

16) (Subset sum) Given an array of non-negative integers and an integer sum. We have to tell

whether there exists any subset in an array whose sum is equal to the given integer sum. Write a dynamic programming algorithm to solve this subset sum problem. Draw the table using array={3,4,5,2] and sum=6.

Examples:

Input: arr[] = {3, 34, 4, 12, 3, 2}, sum = 7

Output: True

Explanation: There is a subset (4, 3) with sum 7.

ANSWER

FUNCTION is\_subset\_sum(arr, target\_sum)

n = LENGTH(arr)

CREATE 2D array dp[n+1][target\_sum+1] initialized to False

// Initialize dp table

FOR i FROM 0 TO n

dp[i][0] = True

// Fill the dp table

FOR i FROM 1 TO n

FOR j FROM 1 TO target\_sum

IF arr[i-1] <= j

dp[i][j] = dp[i-1][j] OR dp[i-1][j-arr[i-1]]

ELSE

dp[i][j] = dp[i-1][j]

RETURN dp[n][target\_sum]

[21:05, 18/05/2024] +90 505 083 77 98: dizi = {3, 4, 5, 2} ve sum = 6 için dinamik tablo örneği

toplam = 0 1 2 3 4 5 6

----------------------------

dizi= []

dp = [ T F F F F F F ]

dizi= [3]

dp = [ T F F T F F F ]

dizi= [3, 4]

dp = [ T F F T T F F ]

dizi= [3, 4, 5]

dp = [ T F F T T T F ]

dizi= [3, 4, 5, 2]

dp = [ T F T T T T T ]

END FUNCTION