EngineerPro - K01

Dynamic Programming

Lam Do

Contents

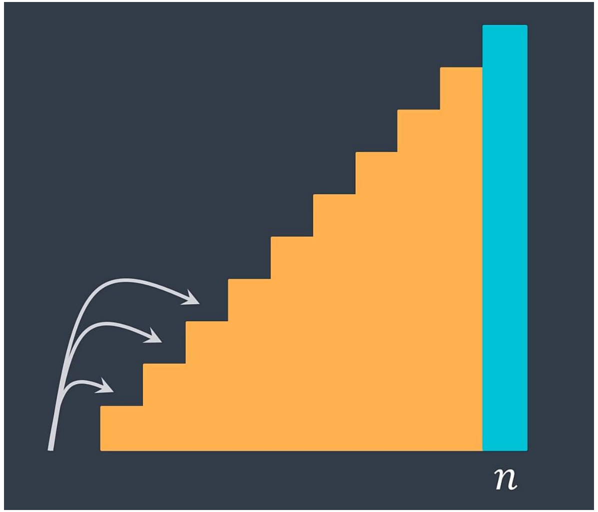
1. Dynamic Programming (DP) Introduction

1. DP Introduction

Problem:

Suppose you have a stairs with n steps, and you want to know the number of ways to reach the nth step (the last one).

Once you jump, you can only jump 1, 2, or 3 steps.

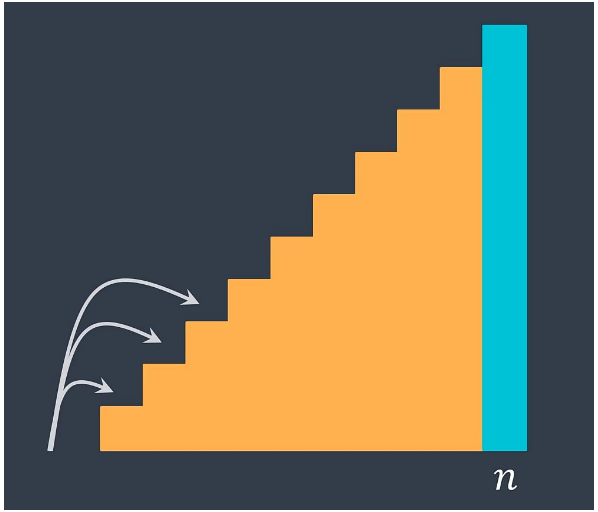


1. DP Introduction

We can approach the problem by calculating the number of way to step to the last one.

jump by 1 step from step n - 1 => a ways
jump by 2 steps from step n - 2 => b ways
jump by 3 steps from step n - 3 => b ways

the ways = a + b + c



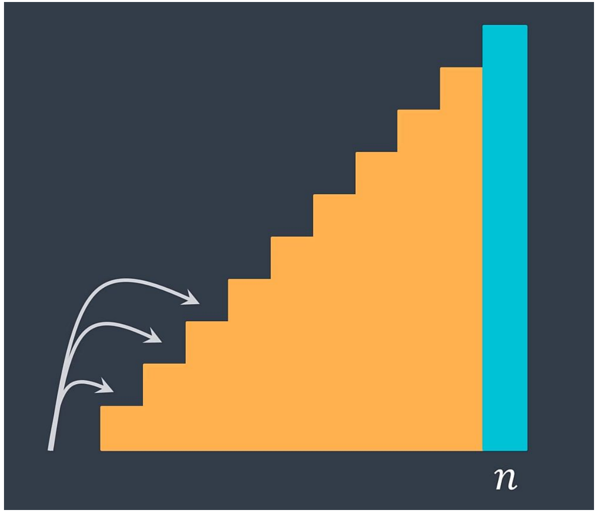
1. DP Introduction

We can approach the problem by calculating the number of way to step to the last one.

jump by 1 step from step n - 1 => a ways
jump by 2 steps from step n - 2 => b ways
jump by 3 steps from step n - 3 => b ways

In other words:
ways(n) = ways(n-1) + ways(n-2) + ways(n-3)

ways(n-1) = ways(n-2) + ways(n-3) + ways(n-4)
ways(n-2) = ways(n-3) + ways(n-4) + ways(n-5)
ways(n-3) = ways(n-4) + ways(n-5) + ways(n-6)



1. DP Introduction

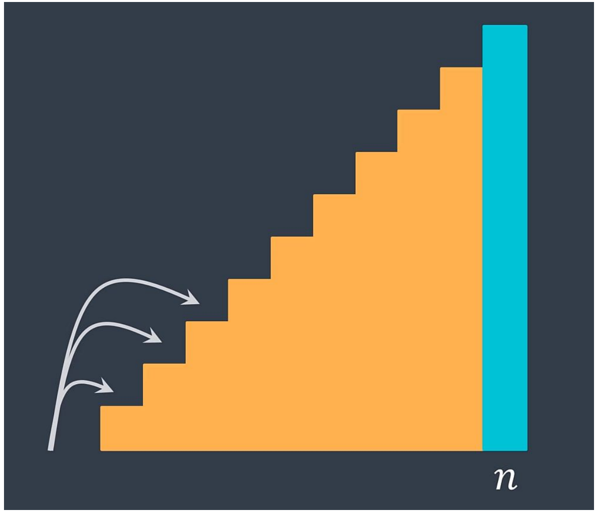
We can approach the problem by calculating the number of way to step to the last one.

jump by 1 step from step n - 1 => a ways
jump by 2 steps from step n - 2 => b ways
jump by 3 steps from step n - 3 => b ways

ways(i) = ways(i-1) + ways(i-2) + ways(i-3)

What if i = 3, 2, 1?

n = 0 -> 1
n = 1 -> 1
n = 2 -> 2
ways(3) = ways(2) + ways(1) + ways(0) = 4
ways(4) = ways(3) + ways(2) + ways(1) = 4 + 2 + 1 = 7



1. DP Introduction

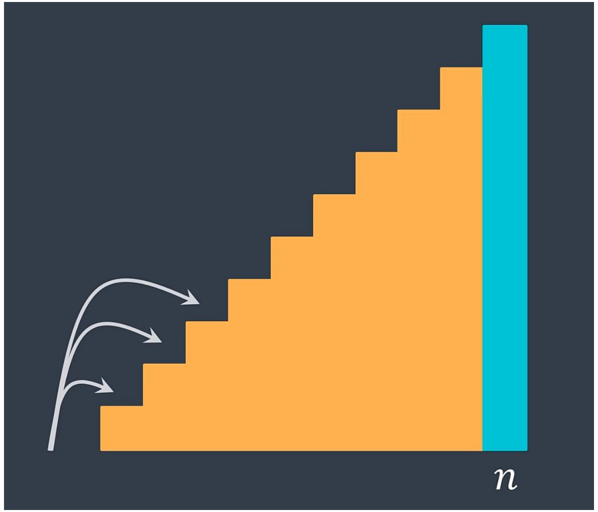
We can approach the problem by calculating the number of way to step to the last one.

jump by 1 step from step n - 1 => a ways
jump by 2 steps from step n - 2 => b ways
jump by 3 steps from step n - 3 => b ways

ways(i) = ways(i-1) + ways(i-2) + ways(i-3)

What if i = 3, 2, 1?

ways(1) = 1
ways(2) = 2
ways(3) = 4

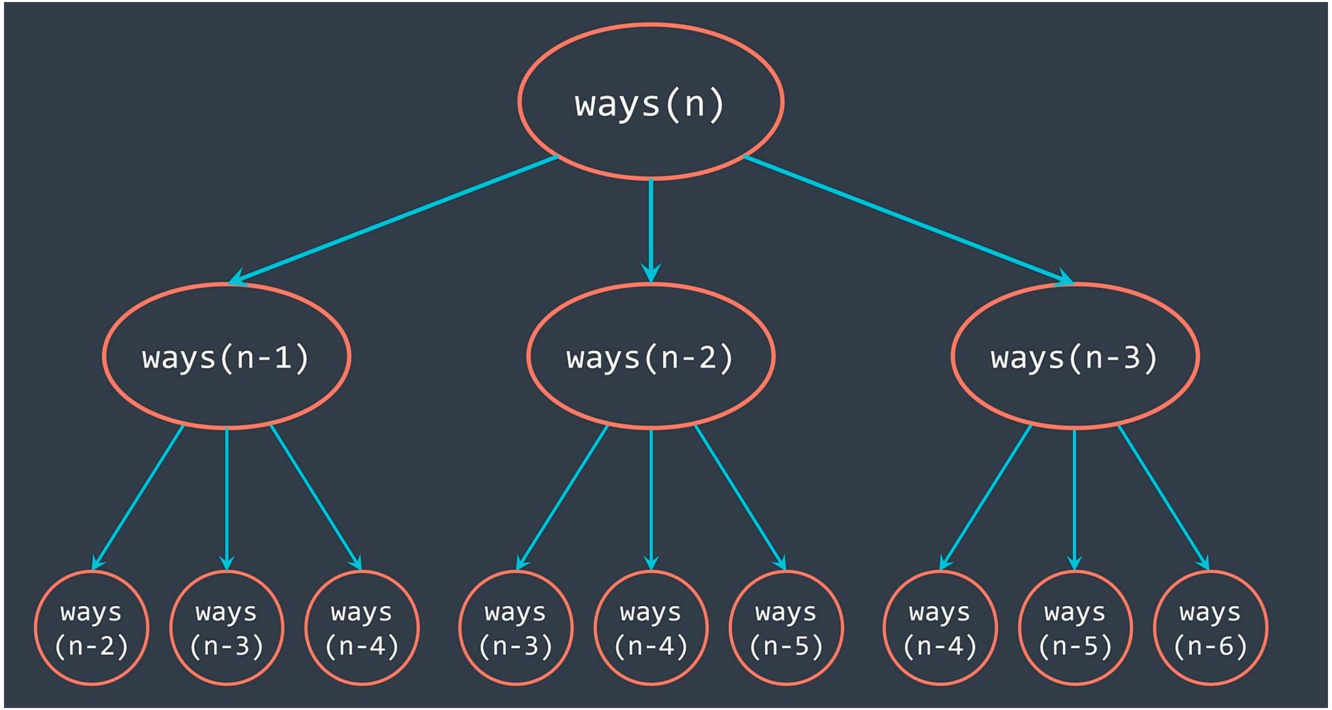


1. DP Introduction

```python
def calculate(x):
    if x == 1:
        return 1
    if x == 2:
        return 2
    if x == 3:
        return 4
    return calculate(x - 1) + calculate(x - 2) + calculate(x - 3)

calculate(n)
```

1. DP Introduction



1. DP Introduction

```python
def calculate(x):
    if x== 1:
        return 1
    if x== 2:
        return 2
    if x== 3:
        return 3
    return calculate(x - 1) + calculate(x - 2) + calculate(x - 3)
```

- Time complexity? O(3^n)
- Space complexity? O(n)

1. DP Introduction

```
def calculate(x):
    if ways[x] > 0:
      return ways[x]
    if x== 1:
       return 1
    if x== 2:
       return 2
    if x== 3:
       return 4
     ways[x]  = calculate(x - 1) + calculate(x - 2) + calculate(x - 3)
    return calculate(x - 1) + calculate(x - 2) + calculate(x - 3)
```

- Time complexity? O(n)
- Space complexity? O(n)

1. DP Introduction

DP Approaches

Top-Down
(memoization)

Bottom-Up
(tabulation)

1. DP Introduction

```
def calculate(n):
    dp = [None * n]
    dp[0] = 1 # at step 1
    dp[1] = 2 # at step 2
    dp[2] = 4 # at step 3
    for i in range(3, n): # step 4 -> step n
        dp[i] = dp[i-1] + dp[i-2] + dp[i-3]
    return dp[n - 1]
```

1. DP Introduction

```
def calculate(n):
      dp = [None * n]
      dp[0] = 1 # at step 1
      dp[1] = 2 # at step 2
      dp[2] = 4 # at step 3
      for i in range(3, n): # i = 3 .. n-1
            dp[i] = dp[i-1] + dp[i-2] + dp[i-3]
      return dp[n-1] # at step n

      broken_stairs = [3, 11, 17] n = 50
```

- Time complexity? O(n)
- Space complexity? O(n)

2. Longest Increasing Subsequence (LIS)

2. Longest Increasing Subsequence (LIS)

Given an array **arr[]** of size **N**, the task is to find the length of the Longest Increasing Subsequence (LIS) i.e., the longest possible subsequence in which the elements of the subsequence are sorted in increasing order.

Input: *arr[] = {3, 10, 12, 1, 20}*
Output: *4*
Explanation: *The longest increasing subsequence is 3, 10, 12, 20*

Input: *arr[] = {3, 2}*
Output:*1*
Explanation: *The longest increasing subsequences are {3} and {2}*

Input: *arr[] = {50, 3, 10, 7, 40, 80}*
Output: *4*
Explanation: *The longest increasing subsequence is {3, 7, 40, 80}*

2. Longest Increasing Subsequence (LIS)

*Let **L(i)** be the length of the LIS **ending at index i** such that arr[i] is the last element of the LIS. Then, L(i) can be recursively written as:*

- *L(i) = max(L(j) + 1) where 0 <= j < i and arr[j] < arr[i]; or*
- ***L(i) = 1**, if no such j exists.*

*Formally, the length of LIS ending at index **i**, is 1 greater than the maximum of lengths of all LIS ending at some index **j** such that **arr[j] < arr[i]** where **j < i**.*

2. Longest Increasing Subsequence (LIS)

Practice:

https://leetcode.com/problems/longest-increasing-subsequence/

https://leetcode.com/problems/longest-increasing-subsequence-ii/

3. Longest Common Subsequence

3. Longest Common Subsequence

Given two strings, **S1** and **S2**, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.

Input: *S1 = "AGGTAB", S2 = "GXTXAYB"*
Output: *4*
Explanation: *The longest subsequence which is present in both strings is "GTAB".*

Input: *S1 = "BD", S2 = "ABCD"*
Output: *2*
Explanation: *The longest subsequence which is present in both strings is "BD".*

3. Longest Common Subsequence

Given two strings, **S1** and **S2**, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.

dp[i][j] = longest common subsequence of i first elements from A and j first elements from B

A = "AGGTAB", *B = "GXTXAYB"*

dp[0][0] = 0 ***dp[0][k] = 0*** ***dp[k][0] = 0***

dp[2][3] = common('AG', 'GXT') = 1 ('G')

dp[0][1] = common('', 'G') = 0 *dp[1][0] = common('A', '') = 0*

dp[0][2] = common('', 'GX') = 0

dp[1][1] = common(A[1], B[1]) = A[1] = B[1] => 1 ("A" , "G") = dp[0][0] + 1

A[1] != B[1] => 0

dp[i][j] = ***dp[i-1][j-1] + (A[i] == B[j])***

A[i] != B[j] => max(dp[i-1][j], dp[i][j-1], ***dp[i-1][j-1]****)* *# dp[i-1][j] = dp[i-2][j] , dp[i-1][j-1]*

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0							
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Creating dp table and filling 0 in 0th row and column

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Updating dp table for row 1

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Updating dp table for row 2

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0							
A	5	0							
B	6	0							

Updating dp table for row 3

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0							

Updating dp table for row 5

3. Longest Common Subsequence

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0	1	1	2	2	3	3	4

Updating dp table for row 6

3. Longest Common Subsequence

Practice:

* https://leetcode.com/problems/longest-common-subsequence/

4. 0/1 Knapsack Problem

4. 0/1 Knapsack problem

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
Output: *3*
Explanation: *There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
Output: *0*

4. 0/1 Knapsack problem

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

dp[i][k] = maximum profit when selecting in i first items, sum weights (selected items) <= k

result = dp[n][W]

dp[0][k] = 0 (0 <= k <= W)

dp[i][0] = 0 (0 <= i <= N)

dp[i][j] = max(

*(select ith item) if j - weight[i] >= 0 -> dp[i-1][**j - weight[i]**] + profit[i]*

(skip ith item) dp[i-1][j]

)

4. 0/1 Knapsack problem

Practice:

https://leetcode.com/problems/partition-equal-subset-sum/description/

https://leetcode.com/problems/number-of-ways-to-earn-points/description/

https://leetcode.com/problems/number-of-great-partitions/description/