# DYNAMIC PROGRAMMING SCRIBE

21st January 2025, Second Half

BHAVYA ARYA

24BM6JP14

## Dynamic programming

**Dynamic programming**, like the divide-and-conquer method solves problems by combining the solutions to sub-problems. Whereas the **Divide and conquer algorithms** partition the problem into disjoint sub-problems **(Decomposition)**, which are then solved **recursively**, and then combine their solutions **(Recomposition)** to solve the original problem. In contrast, **Dynamic programming** applies when the subproblems overlap, that is, when we have multiple identical subproblems to be solved. Dynamic programming algorithm solves each sub-problem just once and then saves its solution through memoization which is a **dynamic programming** technique that stores the results of expensive function calls in a data structures like arrays and tables to avoid redundant calculations. Dynamic programming typically applies to problems such as optimization, where multiple solutions are possible and need to be adjudged

To develop a dynamic-programming algorithm, follow a sequence of four steps:

- Characterize the structure of an optimal solution
- Recursively deduce the value of an optimal solution
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed information

## Edit Distance problem

**Problem Statement**:

Given two strings 'S1' and 'S2'. S1 is to be converted to S2 using following 3 operations:

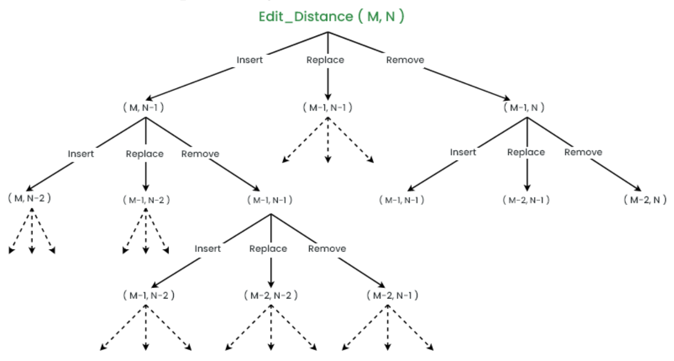
- Deletion of a character.
- Substitution of a character with another one.
- Insertion of a character.

Each operation has a cost corresponding to it and the aim is carry out conversion of string in minimum possible cost.

**Recursive Solution:**

Initial idea is to process all characters one by one, traversing from either end. Each character has two options, either they match or they don't match. If they match we move to the next string, if they don't, we apply all three options and move to the next and recursively formulate for all possible transformations, from this the path with minimum cost is calculated and final solution is obtained

```
FUNCTION EditDistanceRecursive(s1, s2, m, n): # m = len(s1), n=len(s2)
  # Base Cases , if either S1, S2 is empty, add all from  S2 or remove all from s1 respectively
  IF m == 0:
    RETURN n
  IF n == 0:
    RETURN m     # If last characters are the same
  IF s1[m - 1] == s2[n - 1]:
    RETURN EditDistanceRecursive(s1, s2, m - 1, n - 1)
 # If last characters are different, calculate the min of the three
  INSERT = EditDistanceRecursive(s1, s2, m, n - 1)
  REMOVE = EditDistanceRecursive(s1, s2, m - 1, n)
  REPLACE = EditDistanceRecursive(s1, s2, m - 1, n - 1)
  RETURN 1 + MIN(INSERT, REMOVE, REPLACE)
```



Time complexity for this can be obtained recursively as each problem divides into 3 sub-problems $O(3^n)$

**Memoization Solution (Table Approach)**

Taking an example of conversion of string ‘HELLO’(source) to ‘YELLOW’(target). Our aim is to find the minimum cost path for the same. Let the cost for each operation be as following:

- Insertion: +1
- Deletion: +1
- Substitution: +2

Table is initialized with dimensions (n+1) * (m+1), where n is length of the source and m is length of the target. First row represent conversion of an empty string to target string and first column represent conversion of source string to an empty target string. First column and first row will be updated as following:

- For first row, since we are converting an empty string to target string, each character will be inserted, hence cost will be the cost of insertion of each character.
- For first column since we are converting source to an empty string, each character will have to be deleted, hence the cost will be the deletion cost of each character

| | 0 | Y | E | L | L | O | W |
|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **H** | 1 | | | | | | |
| **E** | 2 | | | | | | |
| **L** | 3 | | | | | | |
| **L** | 4 | | | | | | |
| **O** | 5 | | | | | | |

**Step-1: conversion of ‘HE’ to ‘YE’:**

| | |
|---|---|
| ⇩ | Insert E from YE |
| ➡ | Delete E from HE |
| ↘ | Substitute E |

| | 0 | Y | E | L | L | O | W |
|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **H** | 1 | 2 | 3 | | | | |
| **E** | 2 | 3 | 2 | | | | |
| **L** | 3 | | | | | | |
| **L** | 4 | | | | | | |
| **O** | 5 | | | | | | |

**General Formula: At each cell (i, j) in the table:**

- If S1[i] matched S2[j], no transformation is to be performed and cost is carried over from previous diagonal cell.
- If S1[i] is different from S2[j], there are three possible options and minimum cost is computed :
  - Deletion from the source: dp[i-1][j] + deletion cost
  - Insertion into the target: dp[i][j-1] + insertion cost
  - Substitution: dp[i-1][j-1] + substitution cost

| | 0 | Y | E | L | L | O | W |
|---|---|---|---|---|---|---|---|
| **0** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **H** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **E** | 2 | 3 | 2 | 3 | 4 | 5 | 6 |
| **L** | 3 | 4 | 3 | 2 | 3 | 4 | 5 |
| **L** | 4 | 5 | 4 | 3 | 2 | 3 | 4 |
| **O** | 5 | 6 | 5 | 4 | 3 | 2 | 3 |

Final solution is given by te bottom-right cell. Which in this case depicts that the minimum edit distance between HELLO and YELLOW, which is dp[5][6] = 3

Each cell in the table represents the minimum transformation cost for conversion of one string to another using the specified transformations and corresponding costs). Each cell takes constant time, hence the time complexity is **O(m*n)**

# Knapsack (0,1) problem

## Problem Statement

A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The $i^{th}$ item is worth $v_i$ dollars and weighs $w_i$ pounds, where $v_i$ and $w_i$ are integers. Which items should the thief take?
Constraint here is that we can either put an item completely in the bag or cannot put it at all, that is, the value and weight from $i^{th}$ element will either increase by $v_i$ and $w_i$ respectively or none at all.

## Recursion Approach

Simplest solution is to consider all possible subsets having weight less than **W**, and find the subset with highest total value. One of the possible ways of solving this is through recursion following an **inclusion-exclusion** approach, where each item is either selected or not selected progressively to find the total value of items and then the one with maximum value is obtained during recomposition.

```
FUNCTION Knapsack(weights, values, n, W)
  IF n == 0 OR W == 0:
     RETURN 0
  EXCLUDE = Knapsack(weights, values, n - 1, W)
  IF weights[n - 1] <= W:
     INCLUDE = Knapsack(weights, values + $v_{n-1}$, n - 1, W - weights[n - 1])
  ELSE:
     INCLUDE = 0
  RETURN MAX(INCLUDE, EXCLUDE)
```

Time complexity for this can be obtained recursively as:

$$T(n) = 2T(n-1) + 1 = O(2^n)$$

But this can be improved if use **memoization** approach, where we keep on saving the identical sub-problems and avoiding redundant calculations.

## Memoization Approach

For memorization approach, a 2D table can be created in which all values can be stored progressively. Time complexity for which can be obtained recursively as $\boldsymbol{O(n.W)}$where n is the number of items in knapsack and W is its capacity. As there are only n.W values to be calculated.

Taking an example where:
Value, v = {1, 4, 5, 7}
Weights , w = {1, 3, 4, 5}
The knapsack capacity is 7

| Items \ Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 items | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 item (W=1, V=1) | | | | | | | | |
| 2 item (W=3, V=4) | | | | | | | | |
| 3 item (W=4, V=5) | | | | | | | | |
| 4 item (W=5, V=7) | | | | | | | | |

The idea is to build a table(dp) where each entry dp[i][w] will represent the **maximum value** that can be obtained using the first i items for a knapsack of capacity w. We start by creating a table with (n+1) rows and (W+1) rows. Initially, first row is filled with zero as no item gives zero value. Then, iterate through the items and the capacities, filling the table with maximum possible values. For each item i:
For each capacity w (from 1 to the total knapsack capacity), check if including the item gives a better value than excluding it.

Iterations:

- Adding 1st item:

Single value, therefore, it will give highest value as there is no other option

| Items \ Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 item | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 item (W=1, V=1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 item (W=3, V=4) | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
| 3 item (W=4, V=5) | | | | | | | | |
| 4 item (W=5, V=7) | | | | | | | | |

- Adding 2nd item:
  - At capacity = 3, adding Item 2 gives a value of 4, higher than previous value of 1.
  - At capacity = 4, adding Item 2 and item 1 together, gives a value of 5. Similarly for capacity 5,6,7 max values is 5 when both are added

- Adding 3rd item:
  - At capacity = 4, adding Item 3 only gives value 5, which same as previous
  - For capacity = 5,6 Item 3 along with item 1 gives a higher value.
  - For capacity = 7, item 3 long with item 7 gives a higher value.

| Items \ Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 item | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 item (W=1, V=1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 item (W=3, V=4) | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
| 3 item (W=4, V=5) | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
| 4 item (W=5, V=7) | | | | | | | | |

Similarly, adding final item, will give the following result,

Results can be obtained form the bottom right corner of the table where the value is 9 and knapsack is filled at its capacity 7.

| Items \ Capacity | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 item | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 item (W=1, V=1) | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 item (W=3, V=4) | 0 | 1 | 1 | 4 | 5 | 5 | 5 | 5 |
| 3 item (W=4, V=5) | 0 | 1 | 1 | 4 | 5 | 6 | 6 | 9 |
| 4 item (W=5, V=7) | 0 | 1 | 1 | 4 | 5 | 7 | 8 | 9 |

## 2D Water Logging Problem (Homework Problem)

**Problem Statement**

n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining ?

**Possible solutions:**

https://www.geeksforgeeks.org/trapping-rain-water/

## References:

- Edit Distance as on 21st Jan 25: https://www.geeksforgeeks.org/edit-distance-dp-5/
- 0/1 Knapsack problem as on 21st Jan 25: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
- Thomas H. Cormen, Charles E. Lieserson, Ronald L. Rivest and Clifford Stein, ***Introduction to Algorithms***