CSCI 4470 Algorithms

Part IV Advanced Design and Analysis Techniques

- **14 Dynamic Programming**
- 15 Greedy Algorithms
- 16 Amortized Analysis

Chapter 14: Dynamic Programming

```
14 Dynamic Programming
  14.1 Rod cutting
  14.2 Matrix-chain multiplication
  14.3 Elements of dynamic programming
  14.4 Longest common subsequence
  14.5 Optimal binary search trees
```

Rod Cutting Problem

1. Problem Statement:

- Given a rod of length `n` and a list of prices for rods of different lengths, determine the maximum revenue obtainable by cutting the rod into pieces and selling the pieces.

2. Approach:

- Use dynamic programming to break down the problem into smaller, overlapping subproblems.

3. Recursive Solution:

- Define `r(n)` as the maximum revenue for a rod of length `n`.
- Formulate the recursive relation: $r(n) = \max(p_i + r(n-i))$ for $i = 1, 2, ..., n$, where p_i is the price of a rod of length `i`.

4. Dynamic Programming Solution:

- Use a bottom-up approach to build a table of maximum revenues for rods of lengths `1` to `n`.
- For each length `i`, compute the maximum revenue by considering all possible cuts and store the results in the table.

5. Result:

- The table entry for length `n` gives the maximum revenue for a rod of length `n`.

Rod Cutting Example:

Suppose the prices for different lengths of rods are as follows: $p = [1, 5, 8, 9, 10, 17, 17, 20, 24, 30]$

where $p[i]$ is the price of a rod of length $i + 1$. We want to find the maximum revenue for a rod of length 4.

Initialization

1. Create an Array r:

- Start with an array r of length 4, each element initialized to 0: $r = [0, 0, 0, 0]$.

Bottom-Up Calculation

2. For Each Rod Length j:

- For $j = 1$ to 4, perform the following steps.

3. Consider All Possible Cuts:

- For each j, consider all possible cuts i (where i goes from 1 to j).

4. Calculate Maximum Revenue:

- Calculate $q = \max(r[j - i] + p[i - 1])$ for each i and update $r[j]$ if q is greater than the current $r[j]$.

Detailed Calculation

- Given prices $p = [1, 5, 8, 9, 10, 17, 17, 20, 24, 30]$.

5. For $j = 1$:

- Only one cut possible: $q = \max(r[1 - 1] + p[1 - 1]) = \max(r[0] + p[0]) = \max(0 + 1) = 1$.
- So, $r[1] = 1$.

6. For $j = 2$:

- Two possible cuts: $q = \max(r[2 - 1] + p[1 - 1], r[2 - 2] + p[2 - 1]) = \max(r[1] + p[0], r[0] + p[1]) = \max(1 + 1, 0 + 5) = 5$.
- So, $r[2] = 5$.

7. For $j = 3$:

- Three possible cuts: $q = \max(r[3 - 1] + p[1 - 1], r[3 - 2] + p[2 - 1], r[3 - 3] + p[3 - 1]) = \max(r[2] + p[0], r[1] + p[1], r[0] + p[2]) = \max(5 + 1, 1 + 5, 0 + 8) = 8$.
- So, $r[3] = 8$.

8. For $j = 4$:

- Four possible cuts: $q = \max(r[4 - 1] + p[1 - 1], r[4 - 2] + p[2 - 1], r[4 - 3] + p[3 - 1], r[4 - 4] + p[4 - 1]) = \max(r[3] + p[0], r[2] + p[1], r[1] + p[2], r[0] + p[3]) = \max(8 + 1, 5 + 5, 1 + 8, 0 + 9) = 10$.

- So, $r[4] = 10$.

Result

9. Maximum Revenue:

- The maximum revenue for a rod of length 4 is $r[4] = 10$.

Greedy Method

1. Definition:

- The greedy method is an algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

2. Examples:

- Famous examples include the Huffman coding algorithm and Kruskal's algorithm.

3. Pros and Cons:

- Greedy algorithms are simple and often more efficient, but they may not always provide the optimal solution.

Introduction of Dynamic Programming

Definition:

- Dynamic Programming (DP) is a method used to solve complex problems by breaking them into simpler overlapping subproblems.

Application: Primarily used in optimization problems to find the best solution, either by maximizing or minimizing a function.

Characteristics and Importance of Sub-Structure:

- **Overlapping Subproblems**: Subproblems share "sub-subproblems", which differentiates dynamic programming from the divide-and-conquer strategy.
- **Optimal Substructure**: A principle stating that an optimal solution to the entire problem can be constructed from the optimal solutions of its subproblems.

Examples:

- Common DP problems include the Knapsack Problem and Matrix Chain Multiplication.

Memoization & Tabulation:

- DP employs memoization or tabulation techniques to store results of subproblems, preventing redundant calculations.

Advantages of DP over Divide and Conquer:

- Results are stored in a table, allowing retrieval of already calculated solutions.

Steps for Implementing Dynamic Programming:

1. Characterize the Solution Structure:

- `Objective Function` : Define the function that determines what makes a solution optimal.

2. Recursive Definition:

- `Subproblems` : Define the value of the optimal solution in terms of its smaller subproblems.
- `D.P. Recurrence Relation` : Establish a relationship to express the solution of the problem in terms of the solutions of its subproblems.

Notes: ***(D.P. always have the Recurrence Relations)***

3. Compute the Solution

- `Approaches` : Solutions can be computed using either a bottom-up or top-down approach. Memoization is often utilized to store interim values and optimize computation.
  - For a bottom-up approach, build the solution from smaller to larger subproblems using $i - 1$.
  - For a top-down approach, break down the problem from larger to smaller subproblems using $i + 1$.

Example:

- **Items**: Consider a sequence of items, e.g., W_1, W_2, W_3, W_4, W_5.
- **Indexing**: Let i be an item in the sequence. If $W_3 = i$, then $W_2 = i - 1$ and $W_4 = i + 1$.

Selection Table:

- Used in the Bottom-Up Approach.
- Helps to figure out which weights are incorporated in the final solution.

Base Case Explanation:

- The top row in a table represents a base case in dynamic programming.

Pros & Cons:

- DP can yield optimal solutions and enhance efficiency. However, it might have a higher space complexity.

NP-Hard Problems:

- **NP-Hard Problems**: Many NP-hard problems can be formulated as dynamic programming problems, but they often involve a large number of subproblems, making them inefficient.

0-1 Knapsack Problem

0-1 Knapsack Problem Concept

- **Objective**: Maximize the total value of items in the knapsack, ensuring the total weight doesn't exceed the limit.
- **Recursive Function**: Define the solution recursively, evaluating the inclusion of each item.
- **Runtime**: Determined by the table size, $n \times W$, with n as the item number and W as the weight limit.

Solution Process:

1. Objective:

- Identify a set of items M that the knapsack can hold, maximizing total value.

2. Conditions:

- Ensure the total weight $\sum_{x_i \in M} w_i$ in M doesn't exceed the knapsack's total capacity W.
- $\sum_{x_i \in M} w_i \leq W$

3. Maximization:

- Aim to maximize the total value $\sum_{x_i \in M} v_i$ in M.

4. Notations:

- Define: $W_S = \sum_{xi \in S} W_i$ and $V_S = \sum_{xi \in S} V_i$ for total weight and value of set S respectively.
- W_S is used to denote the total weight of a set S where $W_S = \sum_{xi \in S} W_i$.
- V_S is used to denote the total value of a set S where $V_S = \sum_{xi \in S} V_i$.

Conclusion

- **Efficiency**: Dynamic programming ensures efficiency by avoiding redundant computations through memoization.
- **Optimization**: A robust method for tackling optimization and combinatorial issues.

0-1 Knapsack Problem General Method Approach

In the Knapsack Problem, you have a set of items, each with `a weight` and `a value`, and a knapsack that can only carry a limited weight. The goal is to find the most valuable set of items that the knapsack can carry without exceeding the weight limit.

Given:

- Weight limit W of Knapsack.
- Items with weights W_i and values V_i.

Solution:

- The table shows possible solutions for different **weight limits (10, 15, 17)**.

Items	$W = 10$	$W = 15$	$W = 17$
$W_1 = 4, V_1 = 5$	√		√
$W_2 = 5, V_2 = 8$	√	√	√
$W_3 = 7, V_3 = 7$			√
$W_4 = 9, V_4 = 9$		√	

- **For $W = 10$**, Choose Items 1 & 2 ($w = 9, v = 13$).
- **For $W = 15$**, Choose Items 2 & 4 ($w = 14, v = 17$).
- **For $W = 17$**, Choose Items 1, 2 & 3 ($w = 16, v = 20$).

Objective Function:

- Maximize V_S such that S is a subset of I and $W_S \leq W$.
- Objective function: $max\{V_S \mid S \subseteq I \; and \; W_S \leq W\}$

Break down the Objective Function:

Objective: Maximize V_S such that S is a subset of I and $W_S \leq W$.

- I is the set of all items available. In the example, I is the set of all items with their weights and values: $\{(W_1, V_1), (W_2, V_2), (W_3, V_3), (W_4, V_4)\}$.
- S is a subset of items from I that we choose to put in the knapsack. For example, for $W = 17$, S is $\{(W_1, V_1), (W_2, V_2), (W_3, V_3)\}$.
- V_S is the total value of the items in set S. In the example for $W = 17$, V_S is $5 + 8 + 7 = 20$.
- W_S is the total weight of the items in set S. In the example for $W = 17$, W_S is $4 + 5 + 7 = 16$.
- So, the objective function is to maximize the total value V_S of the items in the knapsack, ensuring that the total weight W_S does not exceed the weight limit W.

D.P. Solution for the Knapsack Problem

- **Objective:** Maximize the total value without exceeding the weight limit. start with smaller subproblems and build up solutions progressively.

D.P Recurrence Relation

The recurrence relation for the knapsack problem can be expressed as:

$$Val(W_j, i) = \begin{cases} Val(W_j, i-1) & \text{if } W_j < W_i \\ \max\{Val(W_j, i-1), Val(W_j - W_i, i-1) + V_i\} & \text{if } W_j \geq W_i \end{cases}$$

- If the current item's weight W_i exceeds W_j, it can't be included. So, we only consider the value without this item.
- Otherwise, we choose the maximum value between excluding the item and including it.

Here:

- $Val(W_j, i)$ represents the maximum value for a knapsack of capacity W_j considering the first i items.
- W_i is the weight of the i^{th} item.
- V_i is the value of the i^{th} item.

This relation compares the value of including or excluding item X_i.

- $Val(W, i) = max\{Val(W, i-1), Val(W - W_i, i-1) + V_i\}$

Knapsack Bottom-Up Solution

1. Initialization:
Create tables for weights W_i, values V_i, and items X_i.

- Initialized Table

$W = 15$				
X_i	X_1	X_2	X_3	X_4
W_i	7	4	5	9
V_i	6	5	8	9

- $W = 15$, the capacity of the Knapsack
- X_i, 4 items, W_i the weight of the items, V_i, the values/profits of the items

2. $Val(W_j, i)$ **Calculation:**, to fill up $Val(W_j, i)$ Table

Use the recurrence relation to determine the maximum value for each capacity.

$$Val(W_j, i) = max\{Val(W_j, i-1), Val(W_j - W_i, i-1) + V_i\}$$

- $Val(W, i)$ is the maximum value for a knapsack of capacity W with items $\{X_1, ..., X_i\}$.
- W_i is the weight of the i^{th} item. V_i is the value of the i^{th} item.
- For $W = 15$, the table $Val(W_j, i)$ is filled as shown.

	Wt.	**Val.**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
X_1	7	6	0	0	0	0	0	0	0	6	6	6	6	6	6	6	6	6
X_2	4	5	0	0	0	0	5	5	5	6	6	6	6	11	11	11	11	11
X_3	5	8	0	0	0	0	5	8	8	8	8	13	13	13	14	14	14	14
X_4	9	9	0	0	0	0	5	8	8	8	8	13	13	13	14	14	17	17

- For X_1, $Wt. = 7$, $Val. = 6$. For $W_j = 0$ to 6, $Val(W_j, 1) = 0$. For $W_j = 7$ to 15, $Val(W_j, 1) = 6$.
- For X_2, $Wt. = 4$, $Val. = 5$. For $W_j = 4$ to 6, $Val(W_j, 2) = 5$. For $W_j = 7$ to 15, $Val(W_j, 2) = \max(Val(W_j, 1), Val(W_j - 4, 1) + 5)$.
- For X_3, $Wt. = 5$, $Val. = 8$. For $W_j = 5$, $Val(W_j, 3) = 8$. For $W_j = 6$ to 15, $Val(W_j, 3) = \max(Val(W_j, 2), Val(W_j - 5, 2) + 8)$.
- For X_4, $Wt. = 9$, $Val. = 9$. For $W_j = 9$ to 15, $Val(W_j, 4) = \max(Val(W_j, 3), Val(W_j - 9, 3) + 9)$

3. Construct & Understand $Sel(W_j, i)$ Table:

Use the $Sel(W_j, i)$ table to backtrack and find the items that make up the optimal solution.

When filling the $Val(W, i)$ table, for each entry $Val(W, i)$, compare:

- $Val(W, i - 1)$ (the maximum value without including X_i) $Val(W, i - 1)$
- $Val(W - W_i, i - 1) + V_i$ (the maximum value with including X_i)

If $Val(W, i - 1) < Val(W - W_i, i - 1) + V_i$, set $Sel(W, i) = 1$. Otherwise, set $Sel(W, i) = 0$.

- $Sel(W, i)$ **Table:**

Wt.	Val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	6	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
4	5	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1	1
5	8	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
9	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

- For X_1, $Sel(1, 7) = 1$ because $Val(7, 1) = 6$ is obtained by including X_1.
- For X_2, $Sel(2, 4) = 1$ because $Val(4, 2) = 5$ is obtained by including X_2.
- For X_3, $Sel(3, 5) = 1$ because $Val(5, 3) = 8$ is obtained by including X_3.
- For X_4, $Sel(4, 14) = 1$ because $Val(14, 4) = 17$ is obtained by including X_4.

Backtracking Example:

- $Sel(W, i)$ **Table:**

Wt.	Val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	6	0	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1	1
4	5	0	**0**	0	0	1	1	1	0	0	0	0	1	1	1	1	1
5	8	0	0	0	0	0	1	**1**	1	1	1	1	1	1	1	1	1
9	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**1**

Starting at $Sel(15, 4) = 1$: 從 $Sel(15, 4) = 1$

1. Item 4 (weight 9, value 9) is included. Move up one row and left by 9 columns to $Sel(6, 3)$.
2. At $Sel(6, 3) = 1$, Item 3 (weight 5, value 8) is included. Move up one row and left by 5 columns to $Sel(1, 2)$.
3. At $Sel(1, 2) = 0$, Item 2 is not included. Move up one row to $Sel(1, 1)$.
4. At $Sel(1, 1) = 0$, Item 1 is not included. We've reached the top row, so the process stops.
   Conclusion :

The optimal solution includes items 3 and 4 with a total value of 17.

Indicates if item X_i is in the knapsack for capacity W.

- $Sel(W_j, i) = 1$: Item X_i is in the knapsack.
- $Sel(W_j, i) = 0$: Item X_i is not in the knapsack.
- Once the maximum value is found, use $Sel(W_j, i)$ to construct the optimal solution.
- $Sel(W_j, i)$ tracks items in the knapsack for capacity W_j and items $\{X_1, ..., X_i\}$.

Knapsack Runtime

- **Dynamic Programming Solution**: The knapsack problem's solution using dynamic programming has a runtime of $O(nW)$, where n is the number of items and W is the maximum weight capacity.
- **Table Size**: The table's size is $n \times W$.
- **Time per Entry**: Each table entry is computed in constant time, $\Theta(1)$.
- **Overall Runtime**: The total runtime is $O(nW)$, efficient for small to moderate n and W.
- **Solution Overlap**: Not all solutions for different weight limits overlap.
- **Objective**: Maximize total value within the weight limit.
- **Subproblem Observation**: Some subproblems are subsets of the final solution, but not all.
- **Recurrence Equation**: $Val(W, i)$ represents the best solution for knapsack capacity W with items $\{x\}$.
- **Recursive Function**: $Soln(W, i)$ is the maximum value either without the current item or with it.

Note:

- Solutions for different weight limits may not always overlap.
  - e.g., $W = 10$ is not always part of the solution for weights $W = 15$, an $W = 17$
- The objective is to maximize the total value while staying within the weight limit.
- In dynamic programming, the solution for weight 10 is not always part of the solution for weight 15 or 17.
- It is observed that some subproblems will be a subset of the final solution, but not all. For example, the solution for $W = 10$ is a subset of the solution for $W = 17$, but the solution for $W = 15$ is not a subset of any.
- This requires checking all subproblems and picking the best ones, unlike divide-and-conquer where you can focus on a single subproblem.

Step 1: Start at the Bottom-Right Cell

Begin at the bottom-right cell, $Sel(15, 4)$. This cell represents the optimal solution for a knapsack with a weight capacity of 15, considering all 4 items.

Step 2: Check the Value of the Cell

If $Sel(W, i) = 1$, it means item i is included in the optimal solution. If $Sel(W, i) = 0$, it means item i is not included.

Step 3: Move Up or Left

- If $Sel(W, i) = 1$, move up one row and left by the weight of the included item.
- If $Sel(W, i) = 0$, simply move up one row.

Step 4: Repeat Until the Top Row

Continue this process until you reach the top row.

Knapsack Top-Down Solution

$W = 15$				
X_i	X_1	X_2	X_3	X_4
W_i	7	4	5	9
V_i	6	5	8	9

$Val(W, i) =$ **max value for** $Knapsack(W, \{X_1, ..., X_i\})$:

Wt.	Val.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	6	0		x				x				x	x				x
4	5	0	x					x				x					x
5	8	0						x									x
9	9	0															x

- The First Row usually the base case.
- $Knapsack(15, 4)$ recursive calls
  - $Knapsack(15, 3)$
  - $Knapsack(6, 3)$
- $Knapsack(15, 3)$ recursive calls
  - $Knapsack(15, 2)$
  - $Knapsack(10, 2)$
- $Knapsack(6, 3)$ recursive calls
  - $Knapsack(6, 2)$
  - $Knapsack(1, 2)$
- $Knapsack(15, 2)$ recursive calls

    - $Knapsack(15, 1)$
    - $Knapsack(11, 1)$
- $Knapsack(10, 2)$ recursive calls
    - $Knapsack(10, 1)$
    - $Knapsack(6, 1)$
- $Knapsack(6, 2)$ recursive calls
    - $Knapsack(6, 1)$ (memoized)
    - $Knapsack(2, 1)$
- $Knapsack(1, 2)$ recursive calls
    - $Knapsack(1, 1)$ (memoized)
    - $Knapsack(0, 1)$ (memoized),
      take $max\{0, W - w_i\}$ for first argument

Matrix-chain Multiplication

Chain Operations

EXAMPLE CHAIN OPERATIONS:

Objective: Find the best order to perform a series of operations.

Importance: Vital for optimizing code in compiler design and query optimization in databases.

Matrix Example: Given matrices $A_1, ...A_n$, parenthesize to minimize multiplications, utilizing matrix multiplication's associative property.

(RECALL) Matrix Multiplication:

For $1 \leq i \leq p$ and $1 \leq j \leq r$, the element in the i-th row and j-th column of the resulting matrix C, $C[i, j]$, is computed as the sum of the product of the elements from the i-th row of A and the j-th column of B.

- $1 \leq i \leq p$ and $1 \leq j \leq r$, $C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$

	4×3				3×2			4×2	
$\uparrow$	a_{11}	a_{12}	a_{13}	$\uparrow$	b_{11}	b_{12}	$\uparrow$	c_{11}	c_{12}
p	a_{21}	a_{22}	a_{23}	q	b_{21}	b_{22}	p	c_{21}	c_{22}
$\downarrow$	a_{31}	a_{32}	a_{33}	$\downarrow$	b_{31}	b_{32}	$\downarrow$	c_{31}	c_{32}
	a_{41}	a_{42}	a_{43}					c_{41}	c_{42}
	$\leftarrow q \rightarrow$				$\leftarrow r \rightarrow$			$\leftarrow r \rightarrow$	

Resulting Dimension:

The multiplication of a $p \times q$ matrix with a $q \times r$ matrix results in a $p \times r$ matrix.

- $(p \times q) \times (q \times r) = (p \times r)$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}_{4\times3}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}_{3\times2}, \quad C = AB = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \\ c_{41} & c_{42} \end{bmatrix}_{4\times2}$$

$$C = AB = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} \\ a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} & a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} \end{bmatrix}_{4\times2}$$

How many entries in C?

> The $p \times r$ entries in Matrix C, which of Matrix C $4 \times 2 = 8$ entries

How long to compute each entry? To compute each entry in matrix C from A (4×3) and B (3×2)

> The q multiplications to compute each entry
> Operations, q multiplications and $q - 1$ additions, where $q = 3$.
> Total Operations for Each Entry, $2q - 1$ operations for each entry, so $2 \times 3 - 1 = 5$ operations.

Total time to find Matrix C?

> $p\ q\ r$ multiplications to compute C which is $O(n^3)$
> Time for Each Entry, $2q - 1$ operations, where $q = 3$.
> Total Time Complexity, $p \times r \times (2q - 1)$ operations, $4 \times 2 \times 5 = 40$ operations, equivalent to $O(pqr)$ or $O(n^3)$.

Example of Matrix $A \times B = C$:

For multiplying matrices A (4x3) and B (3x2), resulting in matrix C (4x2), the elements of C would be calculated as:

$$A = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}_{4\times3}, \quad B = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 2 & 0 \end{bmatrix}_{3\times2}, \quad C = AB = \begin{bmatrix} 2 & 2 \\ 3 & 2 \\ 4 & 5 \\ 3 & 3 \end{bmatrix}_{4\times2}$$

$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31}$
$c_{11} = 2 \cdot 0 + 0 \cdot 1 + 1 \cdot 2 = 2$

$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32}$
$c_{12} = 2 \cdot 1 + 0 \cdot 2 + 1 \cdot 0 = 2$

$$C = AB = \begin{bmatrix} c_{11} = (2 \times 0) + (0 \times 1) + (1 \times 2) = 2 & c_{12} = (2 \times 1) + (0 \times 2) + (1 \times 0) = 2 \\ c_{21} = (0 \times 0) + (1 \times 1) + (1 \times 2) = 3 & c_{22} = (0 \times 1) + (1 \times 2) + (1 \times 0) = 2 \\ c_{31} = (1 \times 0) + (2 \times 1) + (1 \times 2) = 4 & c_{32} = (1 \times 1) + (2 \times 2) + (1 \times 0) = 5 \\ c_{41} = (1 \times 0) + (1 \times 1) + (1 \times 2) = 3 & c_{42} = (1 \times 1) + (1 \times 2) + (1 \times 0) = 3 \end{bmatrix}_{4\times2}$$

Matrix Chain Multiplication by DP

Step 1: Understand the Problem

- **Objective**: Minimize the number of scalar multiplications in matrix chain multiplication.

Step 2: Define the Subproblems

- **Notation**: Let $m[i, j]$ denote the minimum number of multiplications needed to compute $A_{i...j}$.

Step 3: Characterize the Structure of an Optimal Solution

- Assume A_i is a $d_{i-1} \times d_i$ matrix for $i = 1, 2, ..., n$.
- Let $A_{i...j}$ be the product of matrices i through j.

Step 4: Find the Recurrence Relation

Recurrence Relation:

- $m[i, j] = \min_{i<k\leq j} \{m[i, k] + m[k + 1, j] + d_{i-1} \times d_k \times d_j\}$

Explanation of d_{i-1}, d_i, and d_j:

- d_{i-1}, d_i, d_j are the dimensions of the matrices involved in the multiplication.
- In the formula, $d_{i-1} \times d_i \times d_j$ calculates the number of scalar multiplications needed to multiply two matrices of dimensions $d_{i-1} \times d_i$ and $d_i \times d_j$.

Step 5: Initialize the Table

- **Initialization**: For each i, $m[i, i] = 0$.

Step 6: Fill in the Table

- **Procedure**: Fill in the table m using the recurrence relation, starting from the smallest subproblems to the largest.

Step 7: Construct the Optimal Solution

- **Solution**: Use the table m to construct the optimal parenthesization.

Step 8: Analyze the Cost

- The cost of the last multiplication $(A_{i..k} \times A_{k+1..j})$ is considered.

Step 9: Consult Previously Evaluated Optimal Sequences

- Each entry in the table is evaluated the same way.

Example Calculation of 3 Matrices

Overview: Demonstrates the impact of multiplication order on the number of operations.

Given: Three matrices with dimensions `5x4` , `4x6` , and `6x2` .

- $m[i,j] = \min_{i<k\leq j}\{m[i,k] + m[k+1,j] + d_{i-1} \times d_k \times d_j\}$

Given Matrices Dimensions: $p = [d_0, d_1, d_2, d_3] = [5, 4, 6, 2]$
For Matrices $A_1(5 \times 4), A_2(4 \times 6), A_3(6 \times 2)$.

A_1	A_2	A_3
5×4	4×6	6×2
$d_0\ d_1$	$d_1\ d_2$	$d_2\ d_3$

$m[i,j]$	1	2	3
1	0	120	88
2	–	0	48
3	–	–	0

$k[i,j]$	1	2	3
1	0	1	1
2	–	0	2
3	–	–	0

Initialization: $m[1,1] = m[2,2] = m[3,3] = 0$. (no multiplications needed for a single matrix)

$$m[1,2] = \min_{1\leq k<2} \begin{cases} k=1: & m[1,1] + m[2,2] + d_0 \times d_1 \times d_2 \\ k=1: & 0 + 0 + 5 \times 3 \times 6 = 120 \end{cases}$$

$$m[2,3] = \min_{2\leq k<3} \begin{cases} k=2: & m[2,2] + m[3,3] + d_1 \times d_2 \times d_3 \\ k=2: & 0 + 0 + 4 \times 6 \times_2 = 48 \end{cases}$$

$$m[1,3] = \min_{1\leq k<3} \begin{cases} k=1: & m[1,1] + m[2,3] + d_0 \times d_1 \times d_3 \\ k=1: & 0 + 48 + 5 \times 4 \times_2 = 88 \\ k=2: & m[1,2] + m[3,3] + d_0 \times d_2 \times d_3 \\ k=2: & 120 + 0 + 5 \times 6 \times_2 = 180 \end{cases}$$

Optimal Parenthesization:

$m = [1,2]\ (A_1)A_2A_3$
$m = [2,3]\ (A_1A_2)A_3$
$m = [1,3]\ (A_1)A_2A_3$

CLASS EXAMPLE:

Objective: Compute the value using Dynamic Programming and construct the optimal solution using $m[i,j]$, and $k[i,j]$.

- Note: $k[i,j]$ keeps track of the placement of parenthesis.

Given: Determine multiplication order for $d =< 3, 9, 8, 2, 5 >$.

Apply the Formula:

$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + d_{i-1} \times d_k \times d_j\}$$

A_1	A_2	A_3	A_4
3×9	9×8	8×2	2×5
$d_0\ d_1$	$d_1\ d_2$	$d_2\ d_3$	$d_3\ d_4$

$m[i,j]$	1	2	3	4
1	0	216	198	228
2	–	0	144	234
3	–	–	0	80
4	–	–	–	0

$k[i,j]$	1	2	3	4
1	0	1	1	3
2	–	0	2	3
3	–	–	0	3
4	–	–	–	0

$$m[1,2] = \min_{1 \le k < 2} \begin{cases} k=1: & m[1,1] + m[2,2] + d_0 \times d_1 \times d_2 \\ k=1: & 0 + 0 + 3 \times 9 \times 8 = 216 \end{cases}$$

$$m[2,3] = \min_{2 \le k < 3} \begin{cases} k=2: & m[2,2] + m[3,3] + d_1 \times d_2 \times d_3 \\ k=2: & 0 + 0 + 9 \times 8 \times_2 = 144 \end{cases}$$

$$m[3,4] = \min_{3 \le k < 4} \begin{cases} k=3: & m[3,3] + m[4,4] + d_2 \times d_3 \times d_4 \\ k=3: & 0 + 0 + 8 \times 2 \times 5 = 80 \end{cases}$$

$$m[1,3] = \min_{1 \le k < 3} \begin{cases} k=1: & m[1,1] + m[2,3] + d_0 \times d_1 \times d_3 \\ k=1: & 0 + 144 + 3 \times 9 \times_2 = 198 \\ k=2: & m[1,2] + m[3,3] + d_0 \times d_2 \times d_3 \\ k=2: & 216 + 0 + 3 \times 8 \times_2 = 264 \end{cases}$$

$$m[2,4] = \min_{2 \le k < 4} \begin{cases} k=2: & m[2,2] + m[3,4] + d_1 \times d_2 \times d_4 \\ k=2: & 0 + 80 + 9 \times 8 \times 5 = 440 \\ k=3: & m[2,3] + m[4,4] + d_1 \times d_3 \times d_4 \\ k=3: & 144 + 0 + 9 \times 2 \times 5 = 234 \end{cases}$$

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} k=1: & m[1,1] + m[2,4] + d_0 \times d_1 \times d_4 \\ k=1: & 0 + 234 + 3 \times 9 \times 5 = 369 \\ k=2: & m[1,2] + m[3,4] + d_0 \times d_2 \times d_4 \\ k=2: & 216 + 80 + 3 \times 8 \times 5 = 416 \\ k=3: & m[1,3] + m[4,4] + d_0 \times d_3 \times d_4 \\ k=3: & 198 + 0 + 3 \times 2 \times 5 = 228 \end{cases}$$

Result: Optimal Parenthesis Placement: $(A_1(A_2A_3))A_4$. 結果: 最優括號放置: $(A_1(A_2A_3))A_4$.

Example of 4 Matrices by D.P. (Video)

Given matrices $A_1(5 \times 4)$, $A_2(4 \times 6)$, $A_3(6 \times 2)$, and $A_4(2 \times 7)$,
The dimensions as below: $p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2, p_4 = 7$

Initialization: $m[1,1] = m[2,2] = m[3,3] = m[4,4] = 0$

A_1	A_2	A_3	A_4
5×4	4×6	6×2	2×7
$p_0\ p_1$	$p_1\ p_2$	$p_2\ p_3$	$p_3\ p_4$

$m[1,2]$ represents the multiplication of $A_1(5 \times 4)$ and $A_2(4 \times 6)$,
$m[1,2] = 5 \times 4 \times 6 = 120$

$m[2,3]$ represents the multiplication of $A_2(4 \times 6)$ and $A_3(6 \times 2)$,
$m[2,3] = 4 \times 6 \times_2 = 48$

$m[3,4]$ represents the multiplication of $A_3(6 \times 2)$ and $A_4(2 \times 7)$,
$m[3,4] = 6 \times_2 \times 7 = 84$

$1 \le k < 2, k = 1$	$2 \le k < 3, k = 2$	$3 \le k < 4, k = 3$
$A_1 \cdot A_2$	$A_2 \cdot A_3$	$A_3 \cdot A_4$
5×4 4×6	4×6 6×2	6×2 2×7
$m[1,2] =$	$m[2,3] =$	$m[3,4] =$
$m[1,1] + m[2,2] + 5 \times 4 \times 6$	$m[2,2] + m[3,3] + 4 \times 6 \times_2$	$m[3,3] + m[4,4] + 6 \times_2 \times 7$
120	48	84

$m[1,3]$ represents the multiplication of $A_1(5 \times 4)$, $A_2(4 \times 6)$ and $A_3(6 \times 2)$,

Two Possibilities for $m[1,3]$ and $2 \le k < 4$:
$A_1 \cdot (A_2 \cdot A_3)$, and $(A_1 \cdot A_2) \cdot A_3$.
$m[1,3] = min\{A_1 \cdot (A_2 \cdot A_3), (A_1 \cdot A_2) \cdot A_3\}$
$m[1,3] = min(88, 180) = 88$

$1 \le k < 3, k = 1$	$1 \le k < 3, k = 2$
$A_1 \cdot (A_2 \cdot A_3)$	$(A_1 \cdot A_2) \cdot A_3$
5×4 (4×6 6×2)	(5×4 4×6) 6×2
$m[1,1] + m[2,3] + 5 \times 4 \times_2$	$m[1,2] + m[3,3] + 5 \times 6 \times_2$
$0 + 48 + 40 = 88$	$120 + 0 + 60 = 180$

Two Possibilities for $m[2,4]$ and $2 \leq k < 4$: $A_2 \cdot (A_3 \cdot A_4)$, and $(A_2 \cdot A_3) \cdot A_4$.

$m[2,4] = min\{A_2 \cdot (A_3 \cdot A_4), (A_2 \cdot A_3) \cdot A_4\}$
$m[2,4] = min(252, 104) = 104$

$2 \leq k < 4, k = 2$	$2 \leq k < 4, k = 3$
$A_2 \cdot (A_3 \cdot A_4)$	$(A_2 \cdot A_3) \cdot A_4$
4×6 (6×2 2×7)	(4×6 6×2) 2×7
$m[2,2] + m[3,4] + 4 \times 6 \times 7$	$m[2,3] + m[4,4] + 4 \times_2 \times 7$
$0 + 84 + 168 = 252$	$48 + 0 + 56 = 104$

Three Possibilities for $m[1,4]$ and $1 \leq k < 4$:
$A_1 \cdot (A_2 \cdot A_3 \cdot A_4)$, $(A_1 \cdot A_2) \cdot (A_3 \cdot A_4)$, and $(A_1 \cdot A_2 \cdot A_3) \cdot A_4$

$k = 1$	$k = 2$	$k = 3$
$A_1 \cdot (A_2 \cdot A_3 \cdot A_4)$	$(A_1 \cdot A_2 \cdot) \cdot (A_3 \cdot A_4)$	$(A_1 \cdot A_2 \cdot A_3) \cdot A_4$
5×4 (4×6 6×2 2×7)	(5×4 4×6) (6×2 2×7)	(5×4 4×6 6×2) 2×7
$m[1,1] + m[2,4] + 5 \times 4 \times 7$	$m[1,2] + m[3,4] + 5 \times 6 \times 7$	$m[1,3] + m[4,4] + 5 \times_2 \times 7$
$0 + 104 + 140 = 244$	$120 + 84 + 210 = 414$	$88 + 0 + 70 = 158$

$m[1,4] =$
$min\{\{m[1,1] + m[2,4] + 5 \times 4 \times 7\}, \{m[1,2] + m[3,4] + 5 \times 6 \times 7\}, \{m[1,3] + m[4,4] + 5 \times 2 \times 7\}\}$
$m[1,4] = min\{244, 414, 158\} = 158$

$m\ i/j$	1	2	3	4
1	0	120	88	158
2	–	0	48	104
3	–	–	0	84
4	–	–	–	0

This table is nothing but the k values, and important for parenthesization for the matrices

s	1	2	3	4
1	0	1	1	3
2	–	0	2	3
3	–	–	0	3
4	–	–	–	0

MATRIX-CHAIN-ORDER Function

```
MATRIX-CHAIN-ORDER(p, n)
1. let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
2. for i = 1 to n       // chain length 1
3.    m[i, i] = 0
4. for l = 2 to n     // l is the chain length
5.    for i = 1 to n − l + 1      // chain begins at A_i
6.        j = i + l − 1        // chain ends at A_j
7.        m[i, j] = ∞
8.        for k = i to j − 1      // try A_{i:k} A_{k+1:j}
9.            q = m[i, k] + m[k + 1, j] + p_{i−1} d_k p_j
10.           if q < m[i, j]
11.               m[i, j] = q     // remember this cost
12.               s[i, j] = k     // remember this index
13. return m and s
```

Runtime $T(n) = O(n^3)$ (Loops nested 3 deep)
Easy to show runtime is actually $O(n^3)$

TOP-DOWN MODIFICATION

Notice there are 3 values that contribute to the k^th iteration computing $m[i, i]$

- $P_{i-1} d_k P_j$
- $m[i, k]$
- $m[k + 1, j]$

If $P_{i-1} d_k P_j < m[i, j]$, stop computation

Similarly, if $P_{i-1} d_k P_j + m[i, k] < m[i, i]$, stop computation

- These branches of computation ill not contribute to the solution … they will only cost time

Edit Distance Problem (No. 15-3)

Problem Statement:

- Given two strings, x and y, determine the minimum number of operations required to transform x into y.

Allowed Operations:

- **Insertion**: Insert a character into x.

- **Deletion**: Delete a character from x.
- **Replacement**: Replace a character in x with another character.

> Each operation has an equal cost of 1.

Recurrence Relation:

- For mismatch at positions `i` of s_1 and `j` of s_2, consider three operations: insert, delete, and replace.
- Recurrence relation is defined as:

$$\mathrm{dp}(i,j) = \begin{cases} \mathrm{dp}(i-1,j-1) & \text{if } s_1[i] = s_2[j] \\ 1 + \min(\mathrm{dp}(i-1,j), \mathrm{dp}(i,j-1), \mathrm{dp}(i-1,j-1)) & \text{if } s_1[i] \neq s_2[j] \end{cases}$$

Where:

- $\mathrm{dp}(i,j)$ is the minimum operations to convert the first i characters of s_1 to the first j characters of s_2.

Edit Distance Problems (From Class)

Edit Distance Example 1:, $s_1 = a\ b\ a\ d$, and $s_2 = a\ b\ a\ c$, covert s_1 to s_2 by only use `Insert` , `Delete` , and `Replace` operations, how many operations will be conducted?

- **One Operation Solution**, The operation is `Replace d` with `c` in the string s_1, That shows one operation. (Minimum number of operation is 1)
- **Two Operations Solution**, The operations are `Delete d` then `Insert c` , the total operations counted 2.

For the strings $s_1 = a\ b\ a\ d$ and $s_2 = a\ b\ a\ c$ the edit distance is 1 because you can `Replace` the last `d` in the string s_1 with `c` to make them identical.

- Another approach `Delete d` , then `Insert c` in the string s_1 which operation is 2.

Edit Distance Problem Solution (Revised)

Edit Distance Example 2: $s_1 = h\ o\ r\ s\ e$, and $s_2 = r\ o\ s$, convert s_1 to s_2.

Given Strings:
$s_1 = \text{”horse”}$
$s_2 = \text{”ros”}$

Table Initialization:
To convert “horse” to “ros”, we’ll create a table where “horse” is represented by the rows and “ros” is represented by the columns. The table will be initialized as:

	" "	r	o	s
" "	0	-2	-4	-6
h	-2	?	?	?
o	-4	?	?	?
r	-6	?	?	?
s	-8	?	?	?
e	-10	?	?	?

Table Filling:

For each cell (i, j):

1. If $s1[i] = s2[j]$, then the value at the cell is the diagonal top-left value.
2. If $s1[i] \neq s2[j]$, then the value at the cell is $-2 + \min(\text{left cell}, \text{top cell}, \text{diagonal top-left cell})$.

After filling the table based on the above rules, the table will look like:

	" "	r	o	s
" "	0	-2	-4	-6
h	-2	-2	-4	-6
o	-4	-4	-2	-4
r	-6	-4	-4	-4
s	-8	-6	-6	-4
e	-10	-8	-8	-6

Explanation:

- The value at the bottom-right cell (`-6`) is the minimum number of operations to convert “horse” to “ros”.
- The path from the top-left to the bottom-right cell represents the operations (insertion, deletion, match/replacement) to convert “horse” to “ros”.

Edit Distance Example 2: $s_1 = h\ o\ r\ s\ e$, and $s_2 = r\ o\ s$, convert s_1 to s_2.

	" "	h	o	r	s	e
" "	0	-2	-4	-6	-8	-10
r	-2	$-$	$-$	$-$	$-$	$-$
o	-4	$-$	$-$	$-$	$-$	$-$
s	-6	$-$	$-$	$-$	$-$	$-$

	" "	h	o	r	s	e
" "	0	-2	-4	-6	-8	-10
r	-2	-1	-1	-3		
o	-4	-1	-1	-1		
s	-8	-3	-2	-1		

Solution 1:

- `Delete e` in the s_1, then match `s` .
- `Delete r` in the s_1, then match `o` .
- `Replace h` with `r` in the s_1.

- It takes 3 operations (matches does not count as operations)

Solution 2:

- `Delete e` in the s_1.
- `Insert o` between `r` and `s` in the s_1.
- `Delete o` between `h` and `r` in the s_1.
- `Delete h` , it takes 4 operations.

Explanation:

- If $s1[i] = s2[j]$, no operation is needed.
- If $s1[i] \neq s2[j]$, perform one operation plus the minimum of three options.

Grid Representation:

- Create a 2D grid where the rows represent characters from s_1 (plus an extra row at the top for initialization) and the columns represent characters from s_2 (plus an extra column at the left for initialization).
- For example, for $s_1 = h\ o\ r\ s\ e$ and $s_2 = r\ o\ s$, the grid will look like this (initially filled with zeros):

```
  0 r o s
0 0 1 2 3
h 1 0 0 0
o 2 0 0 0
r 3 0 0 0
s 4 0 0 0
e 5 0 0 0
```

Filling the Grid:

- Start from the top-left cell (after the initial row and column) and move to the right and downward, filling each cell based on the recurrence relation. The filling should be done row by row from top to bottom and within each row from left to right.
- If $s_1[i] = s_2[j]$, then the value at the cell is the diagonal top-left value. 如果 $s_1[i] = s_2[j]$ ，則單元格的值是對角線左上角的值。
- If $s_1[i] \neq s_2[j]$, then the value at the cell is `1 + min(left cell, top cell, diagonal top-left cell)` .

Path Representation:

- The path from the top-left to the bottom-right of the grid represents the operations.
- A right move represents an insertion, a down move represents a deletion, and a diagonal move represents a match or replacement.

Example

For $s_1 = h\ o\ r\ s\ e$ and $s_2 = r\ o\ s$, after filling the grid based on the recurrence relation, the grid might look like this:

```
  0 r o s
0 0 1 2 3
h 1 1 2 3
o 2 2 1 2
r 3 2 2 2
s 4 3 3 2
e 5 4 4 3
```

In this grid:

- The value at the bottom-right cell (`3`) is the minimum number of operations to convert s_1 to s_2.
- The path from the top-left to the bottom-right cell (`0 -> 1 -> 2 -> 1 -> 2 -> 2 -> 3`) represents the operations (insertion, deletion, match/replacement) to convert s_1 to s_2.

The numbers in the grid represent the minimum number of operations needed to convert a substring of s_1 to a substring of s_2.

- The rows of the grid represent the characters of s_1 and the columns represent the characters of s_2.
- The number in each cell $\mathrm{dp}(i, j)$ represents the minimum number of operations needed to convert the first i characters of s_1 to the first j characters of s_2.

Explanation of the Grid


```
  0 r o s
0 0 1 2 3
h 1 1 2 3
o 2 2 1 2
r 3 2 2 2
s 4 3 3 2
e 5 4 4 3
```


- The top row and the leftmost column are used for initialization.
- The cell at $\mathrm{dp}(0, 0)$ (top-left corner) is `0` , representing no operations needed to convert an empty string to another empty string.
- The cell at $\mathrm{dp}(1, 0)$ is `1` , representing one delete operation needed to convert `h` to an empty string.
- The cell at $\mathrm{dp}(1, 1)$ is `1` , representing one replace operation needed to convert `h` to `r` .
- The cell at $\mathrm{dp}(5, 3)$ (bottom-right corner) is `3` , representing the minimum number of operations needed to convert `horse` to `ros` .
  So, the grid helps in visualizing the dynamic programming approach to solving the problem, showing the minimum number of operations at each step.

Steps to Determine Each Minimum Number of Operations:

1. Initialization:

- Fill the top row with numbers from `0` to `length of s_2` , and the leftmost column with numbers from `0` to `length of s_1` . This represents the cost of converting a string to an empty string.

Example:

```
  0 r o s
0 0 1 2 3
h 1
o 2
r 3
s 4
e 5
```

2. Filling the Grid:

- For each cell $\text{dp}(i, j)$ (excluding the first row and column), compare the characters at `s_1[i-1]` and `s_2[j-1]`.
- If they are equal, the cost is the same as the cost for converting the previous substrings: $\text{dp}(i, j) = \text{dp}(i-1, j-1)$.
- If they are not equal, the cost is `1 + min(\text{dp}(i-1, j), \text{dp}(i, j-1), \text{dp}(i-1, j-1))`, representing the minimum cost of either deleting, inserting, or replacing a character.

Example:

- For cell $\text{dp}(2, 2)$, comparing `h` (from s_1) and `o` (from s_2), they are not equal.
- So, $\text{dp}(2, 2) = 1 + min(\text{dp}(1, 2), \text{dp}(2, 1), \text{dp}(1, 1)) = 1 + min(1, 2, 1) = 2`.

3. Reading the Grid:

- After filling the entire grid, the bottom-right cell $\text{dp}(length\ of\ s_1, length\ of\ s_2)$ will contain the minimum number of operations needed to convert s_1 to s_2.
  Example:
  - The final grid looks like this:
  - The bottom-right cell is `3`, which is the minimum number of operations to convert `horse` to `ros`.

```
  0 r o s
0 0 1 2 3
h 1 1 2 3
o 2 2 1 2
r 3 2 2 2
s 4 3 3 2
e 5 4 4 3
```

This method systematically fills out the grid to find the minimum number of operations needed to convert one string to another.

Bottom-up Approach by Dynamic Programming

Example:

The string $s_1 = ABCAB$, and the string $s_2 = EACB$
The horizontal is `Delete` operation, and the vertical is `Insert` operation
The horizontal is string s_1, and the vertical is string s_2

	s_1	A	B	C	A	B
s_2	0	1	2	3	4	5
E	1	1	2	3	4	5
A	2	1	2	3	3	4
C	3	2	2	2	3	4
B	4	3	2	3	3	3

the solution is ONLY 3 oprations:
Delete A, Delete B, and Insert E, the cost is 3 operations

Explanation of the Grid:

In the given grid, each cell $\mathrm{dp}(i, j)$ represents the minimum number of operations needed to convert the first i characters of s_1 to the first j characters of s_2.

- If $s_1[i] = s_2[j]$, $\mathrm{dp}(i, j) = \mathrm{dp}(i-1, j-1)$.
- If $s_1[i] \neq s_2[j]$, $\mathrm{dp}(i, j) = 1 + \min(\mathrm{dp}(i-1, j), \mathrm{dp}(i, j-1), \mathrm{dp}(i-1, j-1))$.

Analyzing the Given Example:

Looking at the last cell in the grid, $\mathrm{dp}(5, 4) = 3$, which means it takes 3 operations to convert s_1 to s_2.

Steps to Find Operations:

1. **Start at the last cell $\mathrm{dp}(5, 4)$.**
   - Since $\mathrm{dp}(5, 4) = 3$ is not equal to $\mathrm{dp}(4, 3)$, it means an operation is performed.
2. **Move to the cell with the smaller value among $\mathrm{dp}(4, 4)$, $\mathrm{dp}(5, 3)$, and $\mathrm{dp}(4, 3)$.**
   - Move to $\mathrm{dp}(4, 3)$ since it has the smallest value 2 .
3. **Repeat the process until the first cell $\mathrm{dp}(0, 0)$ is reached.**
   - The operations are determined by the path taken.

Result:

The operations are: Delete A, Delete B, and Insert E, with a cost of 3 operations.
This step-by-step approach helps in understanding the operations needed to convert one string to another using the filled grid.

Example:

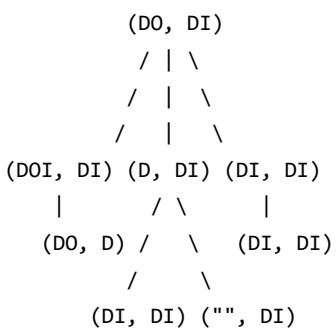
The String $s_1 = DOG$, and the string $s_2 = DIG$

- This example does not use dynamic programming but rather a brute-force recursive approach to explore all possible operations (insert, delete, replace) at each step. Let's go through the example.

Recursive Tree Explanation:

- **Level 0:**
  - Start with the strings s_1 and s_2 as (DO, DI).
- **Level 1:**
  - Explore all possible operations from (DO, DI): insert I (DOI, DI), delete O (D, DI), or replace O with I (DI, DI).
- **Level 2:**
  - Further explore from each state in Level 1: for example, from DOI, DI, delete I (DO, D); from D, DI, insert I (DI, DI) or delete D ("", DI).
- **Level 3:**
  - Continue this process until reaching a state where s_1 is equal to s_2 or no more operations can be performed.

Recursive Tree:



Explanation:

- The tree starts at Level 0 with (DO, DI) and explores all possible operations at each level.
- The leaf nodes represent the final states after all possible operations.
- The goal is to find a path from the root to a leaf where s_1 is equal to s_2 with the minimum number of operations.

The recursive tree represents the operations needed to convert s_1 to s_2. Each level of the tree represents a step in the conversion process.

Recursive Tree:

Level 0:

- Start with the strings $s_1 = DOG$ and $s_2 = DIG$.

Level 1:

- From DOG, DIG, there are three possible operations:
  i. Insert **I** to get DOI, DIG.
  ii. Delete **O** to get DG, DIG.
  iii. Replace **O** with **I** to get DIG, DIG.

Level 2:

- From DOI, DIG:
  i. Delete **I** to get DO, DIG.
  ii. Insert **I** to get $DOII, DIG$.

iii. Replace **I** with **G** to get DOG, DIG.

- From DG, DIG:
  i. Insert **I** to get DGI, DIG.
  ii. Delete **D** to get G, DIG.
  iii. Replace **D** with **I** to get IG, DIG.

Level 3:

- From DO, DIG:
  i. Insert **D** to get DOD, DIG.
  ii. Delete **O** to get DD, DIG.

Does this example use Dynamic Programming with a recursive tree?

For the recursive tree

Level 0 is (DO, DI)

Level 1 (DOI, DI), (D, DI), and (DI, DI)
(DOI, DI) is inserted **I**, (D, DI) is deleted **O**, and (DI, DI) is replaced **O** with **I**.

Level 2 (DO, D), (DI, DI), $(\text{""}, DI)$, (I, DI), and (D, D)
(DO, D) is deleted **I**, (DI, DI) is inserted **I**, $(\text{""}, DI)$ is deleted **D**, (I, DI) is replaced **D** with **I**, and (D, D) is deleted **I**.

Level 3 (DOD, D), and (D, D)
(DOD, D) is inserted **D**, and (D, D) is deleted **O**

please explain this example step by step concisely and comprehensively with minimal wording, please show me the recursive tree with explanations, I have no idea how to do it. please include Taiwan Traditional Chinese after each English statement please
Does this example use Dynamic Programming with recursive tree? please explain this as well.