DESIGN ANALYSIS AND ALGORITHM

Karan Dharmalingam

Deepika C

bcv-ukai-ybw

MEETING ID

bcv-ukai-ybw

IC 1 Access Key

6147

CIA 1 TOPICS

Time Complexity Analysis & Recursion	**Time Complexity Analysis – Mathematical Analysis of Recursive and Non Recursive algorithms**
Searching	Linear, Binary search and Bubble sort
Sorting 1	Insertion sort, Selection sort, quick sort
Sorting 2	Divide and Conquer: mergesort , Binary Heap and Heap sort
Greedy Algorithm	Activity selection problem
Greedy Algorithms	Fractional knapsack,Huffman Coding
String algorithms	Naive algorithm, Rabin Karp algorithm,Z algorithm
String Algorithms	KMP algorithm,Manacher’s algorithm

ACTIVITY SELECTION PROBLEM

Problem Statement

Given n activities with their start and finish times, select the maximum number of activities that can be performed by a single person, assuming that a person can work on only one activity at a time.

Algorithm Explanation

The problem is solved using a Greedy Algorithm. The main idea is to select activities based on their finish times, prioritizing those that finish the earliest, as they leave the maximum room for subsequent activities.

ACTIVITY SELECTION PROBLEM

Steps to Solve

1. Sort Activities by Finish Time
    - Arrange all activities in ascending order of their finish times.
2. Initialize Variables
    - Select the first activity since it finishes earliest.
    - Keep track of the finish time of the last selected activity.
3. Iterate Through Activities
    - For each activity, check if its start time is greater than or equal to the finish time of the last selected activity.
    - If true, select the activity and update the last finish time.
4. Repeat Until End
    - Continue the process for all activities.
5. Output
    - The list of selected activities.

ACTIVITY SELECTION PROBLEM

Detailed Example

Input:

Activities:

- Activity 1: Start = 1, Finish = 2
- Activity 2: Start = 3, Finish = 4
- Activity 3: Start = 0, Finish = 6
- Activity 4: Start = 5, Finish = 7
- Activity 5: Start = 8, Finish = 9
- Activity 6: Start = 5, Finish = 9

ACTIVITY SELECTION PROBLEM

Steps:

1. Sort Activities by Finish Time
2. Sorted activities (based on finish time):
    - Activity 1: (1, 2)
    - Activity 2: (3, 4)
    - Activity 3: (0, 6)
    - Activity 4: (5, 7)
    - Activity 5: (8, 9)
    - Activity 6: (5, 9)
3. Select the First Activity
    - Select Activity 1: (1, 2)
    - Update last finish time: 2.
    - Activity 6: Start = 5 < Last Finish = 9 → Skip.

4. Iterate Through Remaining Activities
    - Activity 2: Start = 3 ≥ Last Finish = 2 → Select.
    - Update last finish time: 4.
    - Activity 3: Start = 0 < Last Finish = 4 → Skip.
    - Activity 4: Start = 5 ≥ Last Finish = 4 → Select.
    - Update last finish time: 7.
    - Activity 5: Start = 8 ≥ Last Finish = 7 → Select.
    - Update last finish time: 9.
    - Activity 6: Start = 5 < Last Finish = 9 → Skip.

5. Selected Activities:
    - Activity 1: (1, 2)
    - Activity 2: (3, 4)
    - Activity 4: (5, 7)
    - Activity 5: (8, 9)

Output:

The maximum number of activities is 4, and the selected activities are:

- (1, 2), (3, 4), (5, 7), (8, 9).

ACTIVITY SELECTION PROBLEM

We are given six activities with their start and finish times:

Activity	Start Time	Finish Time
A1	1	2
A2	3	4
A3	0	6
A4	5	7
A5	8	9
A6	5	9

Step-by-Step Visualization

1. Sort the Activities by Finish Time
    - We arrange the activities in ascending order of their finish time:

A1: (1, 2), A2: (3, 4), A3: (0, 6), A4: (5, 7), A5: (8, 9), A6: (5, 9)

2. Start Selection

- Select A1 because it finishes the earliest.
- Highlight A1:

  Selected: A1 (1, 2)

ACTIVITY SELECTION PROBLEM

3. Iterate Through Remaining Activities

- A2: Start = 3 ≥ Last Finish = 2 → Select A2:

Selected: A1 (1, 2), A2 (3, 4)

- A3: Start = 0 < Last Finish = 4 → Skip A3.
- A4: Start = 5 ≥ Last Finish = 4 → Select A4:

Selected: A1 (1, 2), A2 (3, 4), A4 (5, 7)

- A5: Start = 8 ≥ Last Finish = 7 → Select A5:

Selected: A1 (1, 2), A2 (3, 4), A4 (5, 7), A5 (8, 9)

- A6: Start = 5 < Last Finish = 9 → Skip A6.

ACTIVITY SELECTION PROBLEM

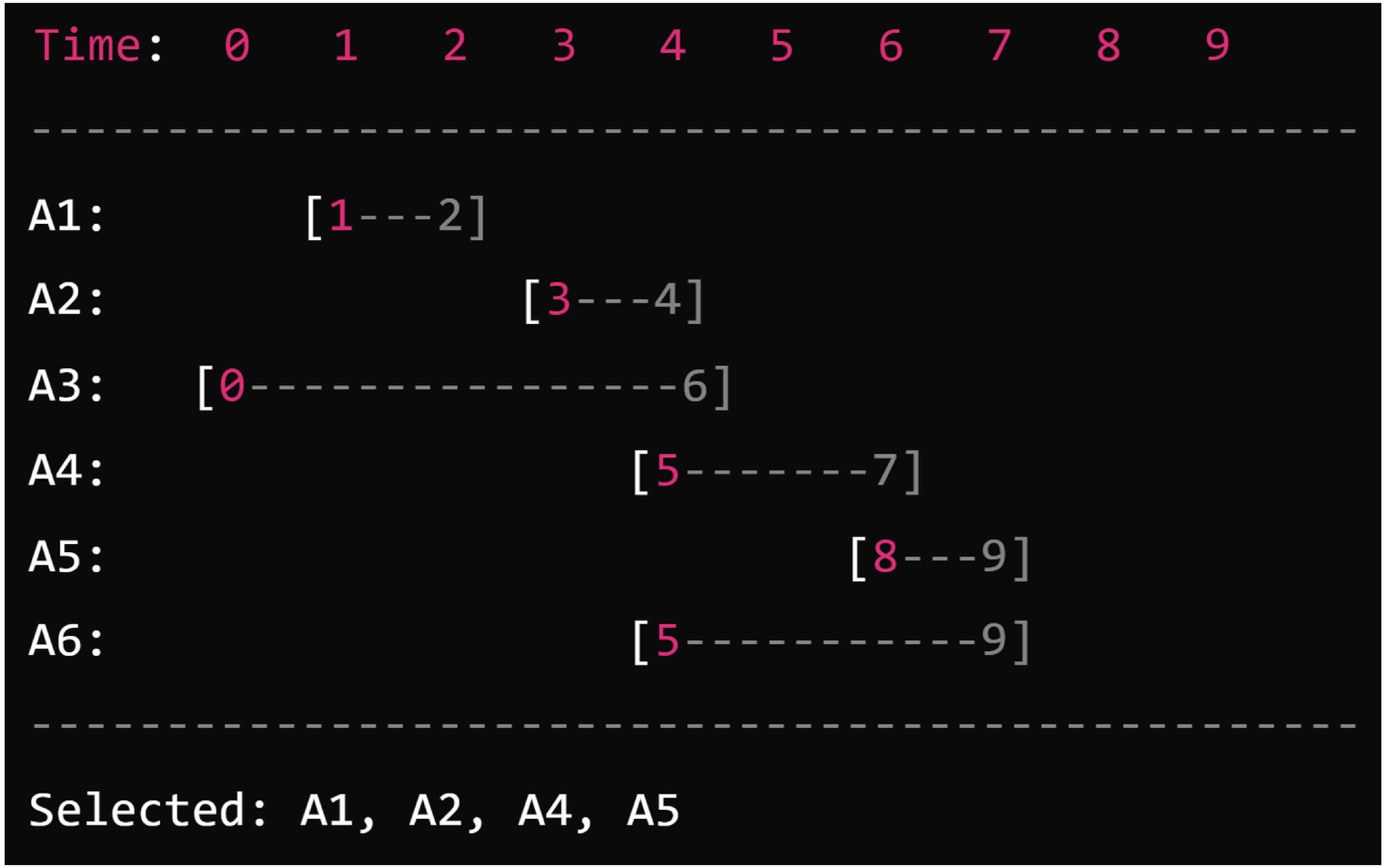
Timeline Visualization

Final Output

The selected activities are:

- A1: (1, 2)
- A2: (3, 4)
- A4: (5, 7)
- A5: (8, 9)

Maximum number of activities: 4.



```
Time:  0   1   2   3   4   5   6   7   8   9
-------------------------------------------------
A1:       [1---2]
A2:               [3---4]
A3:   [0----------------6]
A4:                   [5-------7]
A5:                           [8---9]
A6:                   [5-----------9]
-------------------------------------------------
Selected: A1, A2, A4, A5
```

FRACTIONAL KNAPSACK PROBLEM

Problem Statement

You are given n items, each with a weight (wi) and value (vi). Your goal is to fill a knapsack with a maximum weight capacity (W) to achieve the maximum total value. You can take fractions of an item if its full weight does not fit into the knapsack.

Algorithm Explanation

This problem is solved using the Greedy Algorithm. The key is to maximize the value at each step by choosing items based on their value-to-weight ratio (vi/wi).

FRACTIONAL KNAPSACK PROBLEM

Steps of the Algorithm

1. Compute Value-to-Weight Ratios:
    - Calculate the ratio vi/wi for each item.
2. Sort Items:
    - Sort the items in descending order based on their value-to-weight ratios.
3. Fill the Knapsack:
    - Iterate through the sorted items:
        - If the item's weight fits completely in the remaining capacity, add it to the knapsack.
        - If it doesn't fit, take a fraction of the item that fills the knapsack.
4. Stop When Full:
    - Stop when the knapsack is full or all items have been considered.

FRACTIONAL KNAPSACK PROBLEM

Example

Given Data:

Item	**Weight (w_i)**	**Value (v_i)**	**Value-to-Weight Ratio (v_i / w_i)**
1	10	60	6
2	20	100	5
3	30	120	4

FRACTIONAL KNAPSACK PROBLEM

Steps:

1. **Calculate Ratios**:
   - v1/w1=60/10=6v_1/w_1 = 60/10 = 6v1/w1=60/10=6
   - v2/w2=100/20=5v_2/w_2 = 100/20 = 5v2/w2=100/20=5
   - v3/w3=120/30=4v_3/w_3 = 120/30 = 4v3/w3=120/30=4
2. **Sort by Ratios**:
   - Order: Item 1 → Item 2 → Item 3.

FRACTIONAL KNAPSACK PROBLEM

1. **Fill the Knapsack**:
   - Start with capacity W=50W = 50W=50.
   - **Select Item 1**:
     - Weight: 10 (fits completely).
     - Value: 60.
     - Remaining capacity: 50−10=4050 - 10 = 4050−10=40.
   - **Select Item 2**:
     - Weight: 20 (fits completely).
     - Value: 100.
     - Remaining capacity: 40−20=2040 - 20 = 2040−20=20.
   - **Select a Fraction of Item 3**:
     - Available weight: 20.
     - Fraction: 20/30=2/320/30 = 2/320/30=2/3.
     - Value: (2/3)×120=80(2/3)\times 120 = 80(2/3)×120=80.

FRACTIONAL KNAPSACK PROBLEM

1. Final Output:
   a. Total Value: 60+100+80=24060 + 100 + 80 = 24060+100+80=240.
   b. Items Taken:
      i. Item 1: Full (10/10).
      ii. Item 2: Full (20/20).
      iii. Item 3: 20/3020/3020/30.

FRACTIONAL KNAPSACK PROBLEM

Visualization

Data Representation:

Item	Ratio (vi/wi)	Full Taken	Fraction
Item 1	6	Yes	100%
Item 2	5	Yes	100%
Item 3	4	No	66.67%

Knapsack Timeline:

Initial Capacity: 50

Step 1: Add Item 1 (10/10)

Remaining Capacity: 40

Total Value: 60

Step 2: Add Item 2 (20/20)

Remaining Capacity: 20

Total Value: 160

Step 3: Add 2/3 of Item 3 (20/30)

Remaining Capacity: 0

Total Value: 240

HUFFMAN CODING

Problem Statement

Huffman Coding is an **optimal prefix coding algorithm** used for lossless data compression. The algorithm assigns variable-length binary codes to input characters based on their frequencies, with shorter codes assigned to more frequent characters.

Algorithm Explanation

Huffman Coding uses a **Greedy Algorithm** to build an optimal binary tree (Huffman Tree) from character frequencies. Each character is represented as a leaf in this tree, and binary codes are assigned by traversing the tree.

HUFFMAN CODING

Steps of the Algorithm

1. **Frequency Table**:
    - Count the frequency of each character in the input.
2. **Create Nodes**:
    - Treat each character as a node, with its frequency as the weight.
3. **Build the Huffman Tree**:
    - Place all nodes in a priority queue (min-heap), sorted by frequency.
    - Repeatedly:
        - Remove two nodes with the smallest frequencies.
        - Create a new parent node with a frequency equal to the sum of the two.
        - Add the parent node back to the heap.
    - Continue until one node remains (the root of the Huffman Tree).

- **Generate Codes**:
    - Assign binary codes to each character:
        - Traverse left: Add 0.
        - Traverse right: Add 1.
- **Encode the Input**:
    - Replace each character with its binary code.

HUFFMAN CODING

Example

Input:

- String: "aaabbc"
- Frequency Table:

Character	Frequency
a	3
b	2
c	1

HUFFMAN CODING

Steps:

1. **Create Nodes**:

Initial nodes:

a (3), b (2), c (1)

1. **Build the Huffman Tree**:

Combine nodes c and b:

Node: cb (3)

Combine cb and a:

Root Node: a_cb (6)

Final Tree:

```
 Root (6)
   /  \
  a (3)  cb (3)
         / \
      c (1) b (2)
```

1. **Generate Codes**:

Traverse the tree:

a: 0

c: 10

b: 11

- **Encode the Input**:

Replace each character

with its code:

"aaabbc" → "00011110"

Output

- Encoded String:

  "00011110"
- Character Codes:
  - a: 0
  - b: 11
  - c: 10

HUFFMAN CODING

Visualization

Tree Construction

1. Initial:

[a(3), b(2), c(1)]

2. Combine c and b:

```
[a(3), cb(3)]
  cb
 /  \
c    b
```

3. Combine a and cb:

```
     Root
     (6)
    /   \
a(3) cb(3)
        /  \
     c(1) b(2)
```

Code Assignment

- Traverse the tree:
  - Left child: Add 0.
  - Right child: Add 1.

Encoded String Formation

- Replace characters in the string "aaabbc":

a → 0

b → 11

c → 10

Encoded String: 00011110

NAIVE STRING MATCHING ALGORITHM

Problem Statement

The Naive String Matching Algorithm is a straightforward method for finding all occurrences of a pattern P in a text T. It checks for a match by aligning the pattern at every possible position in the text.

Algorithm Explanation

The algorithm works by sliding the pattern P over the text T one character at a time and checking for matches at each alignment. This is why it’s called "naive"—it does not use any optimization

NAIVE STRING MATCHING ALGORITHM

Steps of the Algorithm

1. **Input**:
   - Text T of length n.
   - Pattern P of length m.
2. **Sliding Window**:
   - Start from the first character of T and align P.
   - Compare characters of P with the corresponding characters in T.
3. **Match Check**:
   - If all characters of P match in the current alignment, record the starting index of the match.
4. **Shift**:
   - Shift P one position to the right and repeat the comparison.
5. **End**:
   - Stop when P can no longer fit into T (i.e., when n–m+1n - m + 1 alignments are done).

NAIVE STRING MATCHING ALGORITHM

Example

Input:

- Text T="AABAACAADAABAAABAA}
- Pattern P="AABA"

Steps:

Alignment 1: Compare T[0..3] with P:

```
Text:    AABAACAADAABAAABAA
Pattern: AABA
Indices: 012345678901234567
```

Match found at index 0.

Alignment 2: Compare T[1..4] with P:

```
Text:    AABAACAADAABAAABAA
          AABA
Indices: 012345678901234567
```

No match.

Alignment 3: Compare T[2..5] with P:

```
Text:    AABAACAADAABAAABAA
           AABA
Indices: 012345678901234567
```

No match.

Alignment 4: Compare T[3..6] with P:

```
Text:    AABAACAADAABAAABAA
            AABA
Indices: 012345678901234567
            ^^^^
```

No match.

Alignment 5: Compare T[4..7] with P:

```
Text:    AABAACAADAABAAABAA
             AABA
Indices: 012345678901234567
             ^^^^
```

No match.

NAIVE STRING MATCHING ALGORITHM

Alignment 6: Compare T[9..12] with P:

```
Text:    AABAACAADAABAAABAA
               AABA
Indices: 012345678901234567
               ^^^^
```

Match found at index 9.

Alignment 7: Compare T[13..16] with P:

```
Text:    AABAACAADAABAAABAA
                  AABA
Indices: 012345678901234567
                  ^^^^
```

Match found at index 13.

Output

The pattern P is found at indices: 0, 9, 13.

NAIVE STRING MATCHING ALGORITHM

Visualization

Text: AABAACAADAABAAABAA

Pattern: AABA

Indices: 012345678901234567

Step 1: Match at index 0

Step 2: No match at index 1

Step 3: No match at index 2

Step 4: No match at index 3

Step 5: No match at index 4

Step 6: Match at index 9

Step 7: Match at index 13

RABIN-KARP ALGORITHM

Problem Statement

Given a text T and a pattern P, find all occurrences of P in T using the Rabin-Karp algorithm. This algorithm leverages a hash function to compare the pattern with substrings of the text.

Algorithm

1. Calculate the hash value of the pattern P.
2. Compute the hash value of the first window of text T.
3. Slide the window across the text:
    - If the hash values of the window and the pattern match, compare the actual characters to confirm the match.
    - Compute the hash value for the next window using a rolling hash technique.
4. Continue until the end of the text is reached.

RABIN-KARP ALGORITHM

Steps to Solve

1. Define a hash function for computing hash values of strings.
2. Calculate the hash of the pattern and the initial window in the text.
3. Slide the pattern window:
    - If the hash matches, check characters to verify.
    - Use a rolling hash to efficiently compute the hash for the next window.
4. Return the indices of all matches.

RABIN-KARP ALGORITHM

Example

Text: GEEKS FOR GEEKS

Pattern: GEEK

Steps:

1. Pattern hash: Compute the hash for GEEK.
2. Compute the initial hash of the first window (GEEKS).
3. Slide the pattern window:
    - Compare hash values. If equal, verify characters.
    - Update the hash using a rolling hash for the next window.
4. Matches occur at indices 0 and 10.

Algorithm Explanation

- Hash Function: Typically, a polynomial hash is used.
- Rolling Hash: Update the hash by adding the new character and removing the old character using modular arithmetic.
- Efficiency: Hash comparisons are constant time, making the average-case complexity O(n+m)O(n + m)O(n+m), where nnn is the text length and mmm is the pattern length.

RABIN-KARP ALGORITHM

Step 1: Define a Hash Function

The core idea of Rabin-Karp is to represent both the pattern and substrings of the text as hash values. Typically, a polynomial rolling hash function is used.

Hash Function

For a string S of length m, its hash value can be computed as:

Hash(S)=(S[0]⋅p0+S[1]⋅p1+…+S[m−1]⋅pm−1)mod q\text{Hash}(S) = (S[0] \cdot p^0 + S[1] \cdot p^1 + \dots + S[m-1] \cdot p^{m-1}) \mod qHash(S)=(S[0]⋅p0+S[1]⋅p1+…+S[m−1]⋅pm−1)modq

Where:

- p is a constant (usually a small prime number, e.g., 31).
- q is a large prime number to avoid overflow and ensure hash uniqueness.

The hash function generates a numerical representation for the string, which will allow quick comparisons.

RABIN-KARP ALGORITHM

Step 2: Calculate the Hash of the Pattern P

We start by computing the hash value of the pattern P that we want to search for in the text T.

For example:

- Pattern: P = "GEEK"
- We calculate its hash value using the above hash formula.

Step 3: Compute the Hash of the First Window in the Text T

Now, we need to compare the pattern P with substrings of the text T. We start by computing the hash value of the first window (substring) of T that is the same length as the pattern.

For example:

- Text: T = "GEEKS FOR GEEKS"
- Initial window: T[0..m-1] = "GEEK"

We calculate the hash for this first window, using the same hash function as we did for the pattern.

RABIN-KARP ALGORITHM

Step 4: Slide the Pattern Window Over the Text

Now, we slide the pattern window one character at a time across the text.

For each new window:

1. We compare the hash values of the current window and the pattern.
2. If the hash values match, we then compare the actual characters to check for a real match (this step handles hash collisions).

RABIN-KARP ALGORITHM

Step 4: Slide the Pattern Window Over the Text

Now, we slide the pattern window one character at a time across the text. For each new window:

1. We compare the hash values of the current window and the pattern.
2. If the hash values match, we then compare the actual characters to check for a real match (this step handles hash collisions).

Z ALGORITHM EXPLANATION

Z Algorithm Explanation

The Z Algorithm is an efficient string-matching algorithm that computes the Z-array for a given string. The Z-array is used to find occurrences of a pattern within a text, and the algorithm runs in O(n) time, where n is the length of the string. The Z-array is a helpful structure because it provides information about the longest common prefix between the pattern and substrings of the text.

Problem Statement

Given a text T and a pattern P, find all occurrences of P in T using the Z algorithm. We concatenate the pattern P and the text T with a special delimiter and then compute the Z-array for the concatenated string. The Z-array for a string S is an array of integers where Z[i] is the length of the longest substring starting from S[i] that is also a prefix of S.

Z ALGORITHM EXPLANATION

Steps to Solve the Problem Using Z Algorithm

Step 1: Concatenate the Pattern and Text

We concatenate the pattern P, a delimiter (which is a character that does not appear in either P or T), and the text T to form a new string S. For example:

$$S=P+\text{delimiter}+T$$

This ensures that we can compute the Z-array in one pass without having to compute the Z-values separately for P and T.

Z ALGORITHM EXPLANATION

```
Start
  z_algo(search_string,pattern)
    concatStr = concatenate pattern + “$” + text
    patLen = length of pattern
    n = length of concatStr

 left = 0 and right = 0

 for i = 1 to n, do
   if i > right, then
     left = i and right = i
     while right < n AND concatStr[right-left]=concatStr[right], do
       increase right by 1
     done
     ZArray[i] = right – left
     decrease right by 1
   else
     k = i – left
     if ZArray[k] < right – i +1, then
       ZArray[i] = ZArray[k]
     else
       left = i
       while right < n AND concatStr[right-left]=concatStr[right], do
         increase right by 1
       done
       ZArray[i] = right – left
       decrease right by 1

 for i = 0 to n – 1, do
   if ZArray[i] = patLen, then
     print the location i – patLen – 1

End
```

Index :	0	1	2	3	4	5	6	7	8	9	10	11	12
Text :	a	a	b	$	c	a	a	b	x	a	a	a	b
Text :	a	a	b	$	c	a	a	b	x	a	a	a	b
Z Values :													

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

The KMP Algorithm is an efficient string-matching algorithm that avoids unnecessary comparisons by using partial matching information. It preprocesses the pattern to create a prefix table (also called the failure function) which is then used to skip over parts of the text that have already been matched.

Problem Statement for KMP

Given a text T and a pattern P, find all occurrences of P in T using the KMP algorithm. The KMP algorithm preprocesses the pattern to avoid re-evaluating characters that have already been compared, leading to faster matching.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Steps to Solve the Problem Using KMP Algorithm

Step 1: Preprocess the Pattern to Create the Prefix Table

The prefix table is an array LPS (Longest Prefix Suffix) where LPS[i] gives the length of the longest proper prefix of the substring P[0..i] that is also a suffix of P[0..i]. The value of LPS[i] indicates how much of the pattern we can skip when a mismatch occurs.

1. Initialize LPS[0] = 0 because a single character has no proper prefix.
2. For each subsequent character in the pattern, check the longest prefix-suffix match and fill in the LPS array.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Step 2: Use the LPS Array to Match the Text

1. Start comparing characters of the text T with the pattern P from the beginning.
2. If a mismatch occurs and the prefix table suggests that part of the pattern has already been matched, shift the pattern accordingly without rechecking the previously matched part.
3. If a match is found, move to the next character of both the text and the pattern.

Step 3: Find Matches

Once a complete match of the pattern with a substring of the text is found, record the index of the match and continue the comparison using the LPS table to skip already checked portions.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

Example Walkthrough for KMP

Text: T = "ABABDABACDABABCABAB"

Pattern: P = "ABABCABAB"

Step 1: Compute the Prefix Table

We compute the LPS (Longest Prefix Suffix) array for the pattern:

1. P[0] = "A": No prefix or suffix, so LPS[0] = 0.
2. P[1] = "B": No match, so LPS[1] = 0.
3. P[2] = "A": Match prefix "A" with suffix, so LPS[2] = 1.
4. P[3] = "B": Match "AB", so LPS[3] = 2.
5. P[4] = "C": No match, so LPS[4] = 0.
6. P[5] = "A": Match "A" again, so LPS[5] = 1.
7. P[6] = "B": Match "AB", so LPS[6] = 2.
8. P[7] = "A": Match "ABA", so LPS[7] = 3.
9. P[8] = "B": Match "ABAB", so LPS[8] = 4.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

So, the LPS array is:

LPS = [0, 0, 1, 2, 0, 1, 2, 3, 4]

Step 2: Match the Pattern with the Text

1. Start comparing the first character of the pattern with the text.
2. If a mismatch occurs, use the LPS array to shift the pattern and skip redundant comparisons.
3. Continue until the pattern matches or the end of the text is reached.

Step 3: Find Matches

We find that the pattern "ABABCABAB" occurs in the text at index 10.

KMP ALGORITHM (KNUTH-MORRIS-PRATT)

KMP Algorithm Visualization

Let’s visualize the KMP matching process for T = "ABABDABACDABABCABAB" and P = "ABABCABAB":

```
Text:      A B A B D A B A C D A B A B C A B A B
Pattern:   A B A B C A B A B
LPS Array: 0 0 1 2 0 1 2 3 4
```

The algorithm compares characters efficiently, skipping unnecessary comparisons using the LPS array.

MANACHER’S ALGORITHM

Manacher's Algorithm is a linear-time algorithm used to find the longest palindromic substring in a given string. It works by expanding palindromes around each center while avoiding redundant calculations using a clever trick called mirror expansion.

Problem Statement for Manacher’s Algorithm

Given a string S, find the longest palindromic substring in linear time using Manacher’s Algorithm.

MANACHER'S ALGORITHM

Steps to Solve the Problem Using Manacher's Algorithm

Step 1: Transform the String

To handle even-length palindromes and simplify the process, we transform the string by inserting a special character (e.g., #) between every character of the string and at the ends. This ensures that we treat both even and odd-length palindromes uniformly.

For example, transform S = "abac" into T = "#a#b#a#c#".

Step 2: Initialize Arrays

1. P array: The P[i] array stores the length of the palindrome centered at index i in the transformed string.
2. C: The center of the rightmost palindrome found so far.
3. R: The right edge of the rightmost palindrome found so far.

MANACHER’S ALGORITHM

Step 3: Expand Around Centers

Iterate through the transformed string. For each position i, attempt to expand the palindrome centered at i by comparing characters symmetrically around i. If you reach a mismatch or the bounds of the string, stop.

If a palindrome at i extends past the right boundary R, update C and R.

Step 4: Use Mirror Expansion

For each position i, we check the mirror of i relative to C (i.e., mirror = 2*C - i). If the palindrome at mirror is within the bounds of the current palindrome, we can use it to avoid redundant checks.

Step 5: Find the Longest Palindrome

The maximum value in the P array gives the length of the longest palindromic substring.

MANACHER’S ALGORITHM

Example Walkthrough for Manacher’s Algorithm

Input String: S = "babad"

Step 1: Transform the String

We transform the string S = "babad" into:

T = "#b#a#b#a#d#"

Step 2: Initialize Arrays

- P = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
- C = 0
- R = 0

-

Step 3: Expand Around Centers

- Start with i = 1 (center a):
  - Compare T[0] and T[2], they are equal, so P[1] = 1.
- Move to i = 2 (center b):
  - Compare T[1] and T[3], they are equal, so P[2] = 1.
- Continue expanding and updating P.

Step 4: Find Longest Palindrome

At the end of the expansion, the largest value in P will correspond to the longest palindrome. In this case, the longest palindrome is "bab" (length 3), centered at index 2.

MANACHER'S ALGORITHM

Manacher's Algorithm Visualization

Input: b a b a d

Transformed: # b # a # b # a # d #

P array: [0, 1, 1, 3, 1, 1, 3, 1, 1, 1, 0]

Here, the palindrome of maximum length is "bab" with a center at position 2.

BACKTRACKING

Backtracking is a recursive algorithmic technique used to solve problems by trying all possible solutions and undoing (backtracking) when a solution fails. It is commonly used for constraint satisfaction problems like Sudoku, N-Queens, Maze Solving, and Graph Coloring.

Key Steps in Backtracking

1. Choose – Select a potential solution.
2. Explore – Recursively attempt different possibilities.
3. Backtrack – If a choice leads to failure, undo it and try another option.

Example: Rat in a Maze (Backtracking)

- The rat moves in all possible directions (right, down, left, up).
- If it reaches a dead end, it backtracks and tries another path.

BACKTRACKING VS. GREEDY VS. BRUTE FORCE

Feature	Backtracking	Greedy Algorithm	Brute Force
Approach	Tries all possible solutions but undoes wrong choices (recursive trial-and-error)	Picks the best possible choice at each step, without revisiting	Tries every possible solution without optimization
Use Cases	Problems with constraints (e.g., N-Queens, Maze solving, Sudoku)	Optimization problems (e.g., Dijkstra’s Algorithm, Huffman Coding)	Small problems where all solutions can be checked (e.g., checking all subsets)
Efficiency	More optimized than brute force (prunes incorrect paths)	Most efficient when a greedy choice leads to an optimal solution	Least efficient, as it tries all possibilities
Completeness	Guarantees an optimal solution if implemented correctly	May not always give the optimal solution	Always finds the optimal solution but is very slow
Time Complexity	Generally O(2^N) or O(N!)	Usually O(N log N) or O(N)	Worst-case O(N!) or O(2^N)
Memory Usage	Uses recursion stack (DFS approach)	Low memory, as it doesn’t store past choices	High memory, as it stores all possibilities

RAT IN A MAZE PROBLEM (BACKTRACKING APPROACH)

The Rat in a Maze problem is a classic backtracking problem where a rat starts at the top-left corner of a maze and must reach the bottom-right corner. The maze consists of open paths (represented as 1) and blocked paths (represented as 0). The rat can only move in specific directions (usually down, right, up, and left). The goal is to find a path from start to finish using backtracking.

Steps to Solve the Problem

1. Start from the top-left cell (0,0).
2. If the current cell is the destination (n-1, n-1), print/store the solution.
3. Check if the current cell is a valid move (inside the maze and not blocked).
4. Mark the current cell as visited.
5. Move in possible directions (right, down, left, up) recursively.
6. If a move does not lead to a solution, backtrack (unmark the cell and try another path).
7. If no path exists, return false.

VISUALIZATION OF THE MAZE

Example Input Maze (5x5)

```
1 0 0 0 0
1 1 1 1 0
0 1 0 1 0
0 1 0 1 1
0 1 1 1 1
```

Output Path (if exists)

```
1 0 0 0 0
1 1 1 1 0
0 0 0 1 0
0 0 0 1 1
0 0 0 0 1
```

Where 1 represents the path taken by the rat.

VISUALIZATION OF THE MAZE

1	0	0	0
1	1	0	1
0	1	1	1
0	0	0	1

We represent movement as:

D → Down

R → Right

U → Up

L → Left

VISUALIZATION OF THE MAZE

1	0	0	0
1	1	0	1
0	1	1	1
0	0	0	1

Step	***Position (x, y)***	***Path So Far***	***Decision***	***Next Step***
1	(0,0)	"" (Start)	Can move D	Move to (1,0)
2	(1,0)	D	Can move D, R	Move to (2,0) (Blocked) → Backtrack
3	(1,0)	D	Can move R	Move to (1,1)
4	(1,1)	DR	Can move D, R	Move to (2,1)
5	(2,1)	DRD	Can move D, R	Move to (2,2)
6	(2,2)	DRDR	Can move D, R	Move to (2,3)
7	(2,3)	DRDRR	Can move D, L	Move to (3,3) (Destination reached ✅)

SUBSET SUM PROBLEM USING BACKTRACKING

Problem Statement:

Given a set of N positive integers and a target sum S, determine whether there exists a subset whose sum is equal to S.

For example:

Input:

Set = {3, 34, 4, 12, 5, 2}

Target sum = 9

Output: Yes (Subset {4, 5} or {3, 2, 4} gives sum 9)

SUBSET SUM PROBLEM USING BACKTRACKING

Approach using Backtracking

Key Idea:

- Explore all possible subsets.
- If the sum exceeds S, backtrack (i.e., stop exploring that path).
- If the sum matches S, return success.

Steps for Backtracking Algorithm:

1. Start from the first element.
2. Either include or exclude the current element in the subset.
3. If the sum of the current subset exceeds S, backtrack.
4. If a subset-sum equals S, return true.
5. Repeat recursively for all elements.

Example Input:

- Set: {3, 34, 4, 12, 5, 2}
- Target Sum: 9

Step	***Subset (Path)***	***Remaining Sum (target)***	***Decision***	***Next Step***
1	{}	9	Start	Try including 3
2	{3}	9 - 3 = 6	Include 3	Try including 34
3	{3, 34}	6 - 34 = -28 (Invalid)	Backtrack	Exclude 34
4	{3}	6	Exclude 34	Try including 4
5	{3, 4}	6 - 4 = 2	Include 4	Try including 12
6	{3, 4, 12}	2 - 12 = -10 (Invalid)	Backtrack	Exclude 12
7	{3, 4}	2	Exclude 12	Try including 5
8	{3, 4, 5}	2 - 5 = -3 (Invalid)	Backtrack	Exclude 5
9	{3, 4}	2	Exclude 5	Try including 2
10	{3, 4, 2}	2 - 2 = 0 ✅	Target Reached!	✅ Solution Found

N-QUEENS PROBLEM USING BACKTRACKING

Problem Statement

The N-Queens problem is a classic combinatorial problem where we need to place N queens on an N × N chessboard such that no two queens attack each other.

A queen can attack another queen if:

1. They are in the same row.
2. They are in the same column.
3. They are on the same diagonal.

The goal is to find all possible ways to place N queens on the board while following these constraints.

N-QUEENS PROBLEM USING BACKTRACKING

Algorithm to Solve the N-Queens Problem Using Backtracking

Backtracking is used to explore all possible placements of queens and discard invalid configurations.

Steps to Solve

1. Start placing queens row by row. Begin with the first row and try placing a queen in each column.
2. Check if the placement is safe. Ensure that no other queen threatens the current position.
3. If the position is valid, place the queen and move to the next row.
4. If all queens are placed successfully, store the solution. If not, backtrack by removing the last placed queen and trying a different column.
5. Continue until all solutions are found.

N-QUEENS PROBLEM USING BACKTRACKING

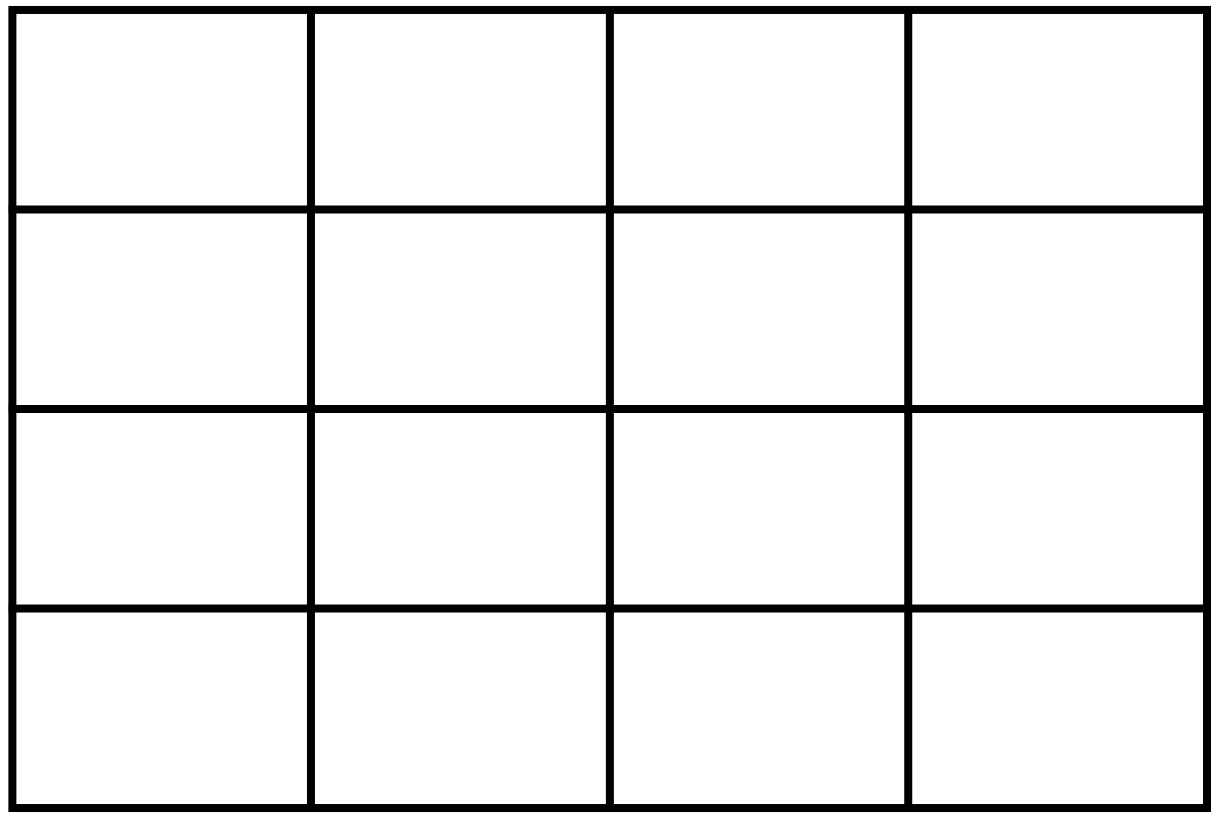
1. Sure! Below is a step-by-step breakdown of the N-Queens problem with visualization for N = 4 using backtracking.

Step-by-Step Process (N = 4)

We will place 4 queens on a 4×4 chessboard such that no two queens attack each other.

Step 1: Start with an Empty Board

A . represents an empty cell, and a Q represents a queen.



N-QUEENS PROBLEM USING BACKTRACKING

Step 2: Place the First Queen in Row 0

We start by placing the first queen (Q1) in column 0 of row 0.

Q			

N-QUEENS PROBLEM USING BACKTRACKING

Step 3: Place the Second Queen in Row 1

- Column 0 is attacked (same column as Q1).
- Column 1 is attacked (diagonal from Q1).
- Column 2 is safe → Place Q2 in (1,2).

Q			
		Q	

N-QUEENS PROBLEM USING BACKTRACKING

Step 4: Place the Third Queen in Row 2

- Column 0 is safe → Place Q3 in (2,0).

Q			
		Q	
Q			

- BUT, this will lead to no valid placement for the 4th queen.
- Backtrack by removing Q3 and trying the next column.

Q			
		Q	
	Q		

- Column 1 is attacked (diagonal from Q2).
- Column 2 is attacked (same column as Q2).
- Q2 is at (1,2).
- Q3 at (2,3) is attacked diagonally by Q2.

Q			
		Q	
			Q

N-QUEENS PROBLEM USING BACKTRACKING

Checking available columns for Q3 in row 2:

- Column 0: Safe ✅
- Column 1: Attacked by Q2 ❌
- Column 2: Attacked (same column as Q2) ❌
- Column 3: Attacked (diagonal from Q2) ❌
- BACKTRACK to Q2 and move it.

Q			
			Q

- Start placing Q3,
- Col 0 - Q1 will attack ❌
- Move to Col 1✅

Q			
			Q
	Q		

- Start placing Q4,
- Col 0 - Q1 will attack❌
- Col 1 - Q3 will attack❌
- Col 2 - Q3 will attack❌
- Col 3 - Q2 will attack❌

Q			
			Q
	Q		
			Q

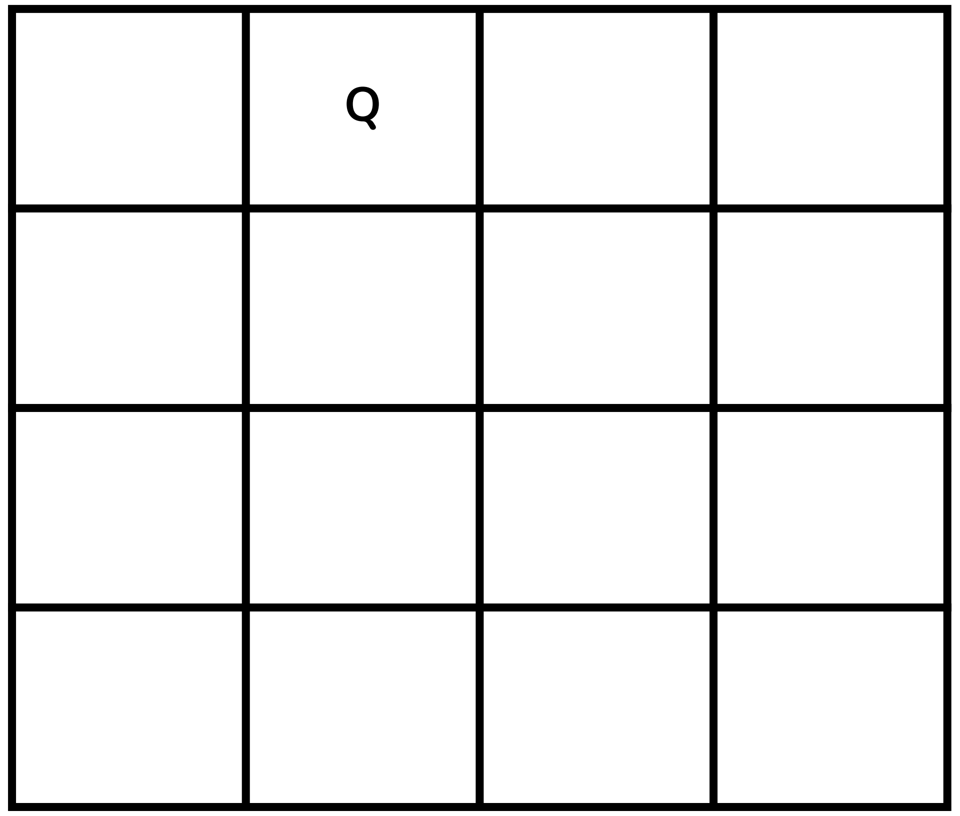
N-QUEENS PROBLEM USING BACKTRACKING

Backtrack to Q3 and move it,

- Col 2 is unsafe ❌
- Col 3 is unsafe ❌

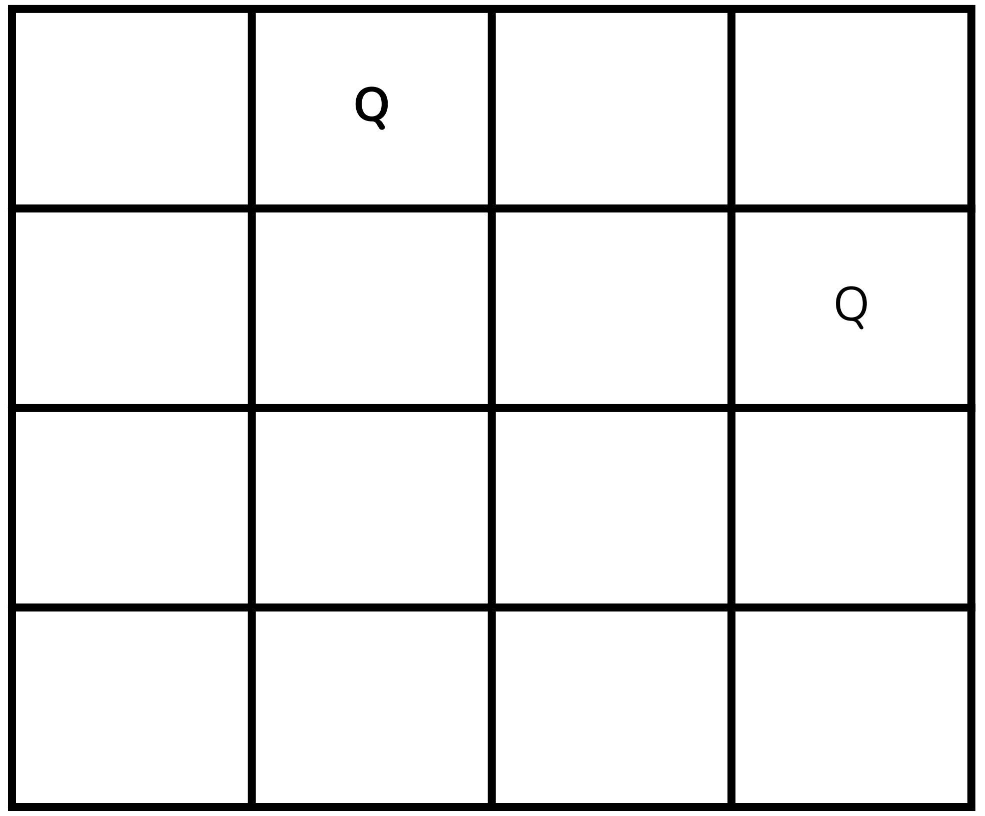
Backtrack to Q2,

- No places left,
- Backtrack to Q1 and move



Start placing Q2,

- Col 0 - Q1 will attack ❌
- Col 1 - Q1 will attack ❌
- Col 2 - Q1 will attack ❌
- Col 3 - Safe ✅

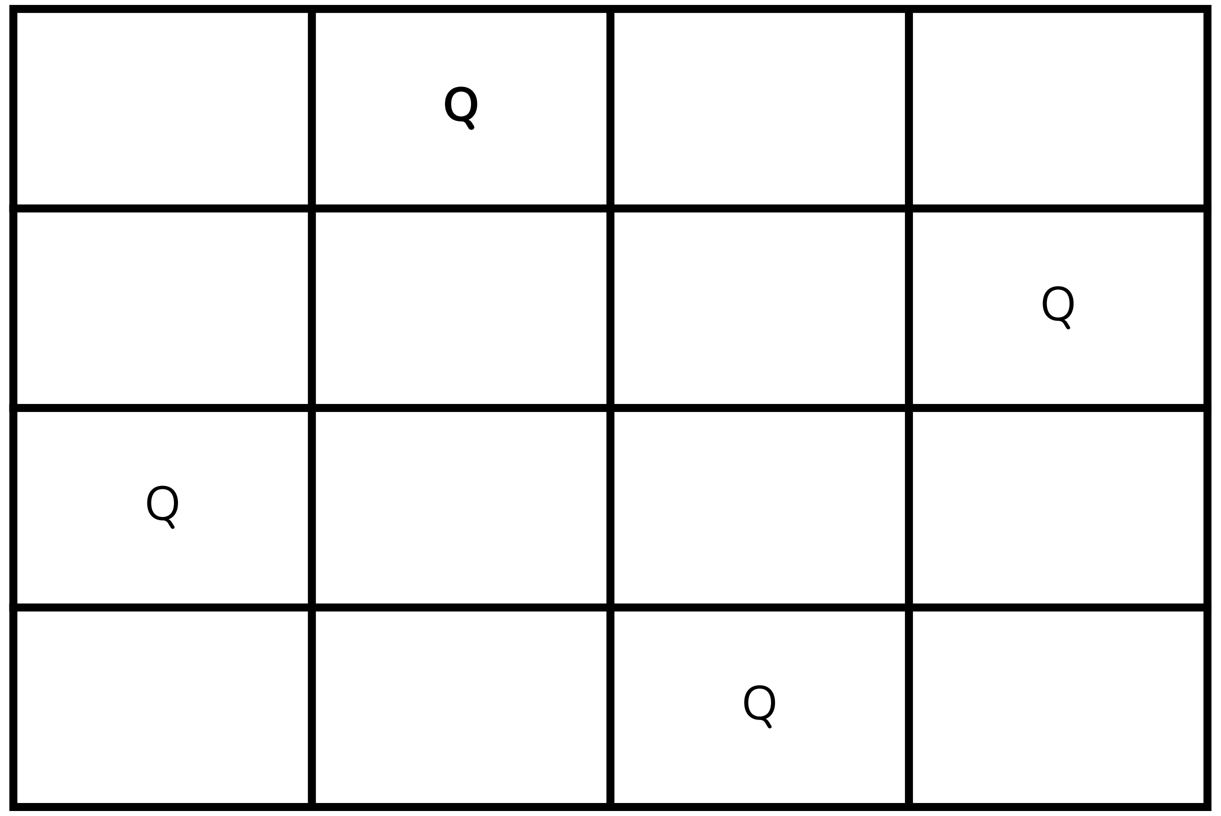


Start placing Q3,

- Col 0 - Safe ✅

Start placing Q4

- Col 0 - Q3 will attack❌
- Col 1 - Q1 will attack❌
- Col 3 - Safe ✅



PRIME NUMBERS AFTER N WITH SUMS

Problem Statement

Given an integer N, find all prime numbers greater than N and compute their sum SumS.

Algorithm

1. Start from (N + 1).
2. Check if the number is prime.
3. If prime, add it to the sum and recursively check the next number.
4. Backtrack:
    - If adding a number doesn't lead to a valid prime sequence (e.g., reaching a count limit), remove it and try the next candidate.
5. Stop when a condition is met (e.g., finding K primes or reaching a number limit).

PRIME NUMBERS AFTER N WITH SUMS

Algorithm

1. Start from (N + 1).
2. Check if the number is prime.
3. If prime, add it to the sum and recursively check the next number.
4. Backtrack:
    - If adding a number doesn't lead to a valid prime sequence (e.g., reaching a count limit), remove it and try the next candidate.
5. Stop when a condition is met (e.g., finding K primes or reaching a number limit).

PRIME NUMBERS AFTER N WITH SUMS

Example

- Input: N = 10, K = 5
- Finding: First 5 prime numbers after 10 and their sum.

Step	Current Number	Is Prime?	Sum So Far	Action
1	11	✅ Yes	11	Add to sum
2	12	❌ No	11	Skip
3	13	✅ Yes	24	Add to sum
4	14	❌ No	24	Skip

PRIME NUMBERS AFTER N WITH SUMS

Example

- Input: N = 10, K = 5
- Finding: First 5 prime numbers after 10 and their sum.

Step	Current Number	Is Prime?	Sum So Far	Action
5	15	❌ No	24	Skip
6	16	❌ No	24	Skip
7	17	✅ Yes	41	Add to sum
8	18	❌ No	41	Skip

PRIME NUMBERS AFTER N WITH SUMS

Example

- Input: N = 10, K = 5
- Finding: First 5 prime numbers after 10 and their sum.

Final Sum: 83

Primes found: 11, 13, 17, 19, 23

Sum: 11 + 13 + 17 + 19 + 23 = 83

Step	Current Number	Is Prime?	Sum So Far	Action
9	19	✅ Yes	60	Add to sum
10	20	❌ No	60	Skip
11	21	❌ No	60	Skip
12	22	❌ No	60	Skip
13	23	✅ Yes	83	Add to sum (STOP)

SIEVE OF SUNDARAM USING BACKTRACKING

Problem Statement

The Sieve of Sundaram is a method to find all prime numbers up to a given limit 2N + 2. Instead of iterating over numbers, we use backtracking to mark numbers that are not prime.

How Does It Work?

1. Remove numbers that follow this formula: **$i+j+2ij \le N$**
2. where i and j are positive integers and $i \le j$.
3. The remaining numbers in the list are converted into primes using the formula: **$2X+1$**
4. where X is an unmarked number.

SIEVE OF SUNDARAM USING BACKTRACKING

Problem Statement

The Sieve of Sundaram is a method to find all prime numbers up to a given limit 2N + 2. Instead of iterating over numbers, we use backtracking to mark numbers that are not prime.

How Does It Work?

1. Remove numbers that follow this formula:
2. $i+j+2ij \leq N$
3. where i and j are positive integers and $i \leq j$.
4. The remaining numbers in the list are converted into primes using the formula:
5. $2X+1$
6. where X is an unmarked number.

SIEVE OF SUNDARAM USING BACKTRACKING

Algorithm Using Backtracking (Step-by-Step)

Instead of iterating, we use recursion to mark numbers.

Step 1: Create a list

- Create an array of size N+1, initially all False (meaning all numbers are "potential primes").

Step 2: Use Backtracking to Mark Non-Primes

- Start with i = 1, j = i, and compute: i+j+2ij
- If the computed value is ≤ N, mark it as True (not prime).
- If the value exceeds N, backtrack (go back and change i or j).

Step 3: Convert Remaining Numbers to Primes

- For every unmarked number X, convert it using: 2X+1
- The result is a prime number.

SIEVE OF SUNDARAM USING BACKTRACKING

Why This Formula Works?

The formula originates from a mathematical property of odd composite numbers. Any odd composite number can be represented in the form:

$2n+1=(2i+1)\times(2j+1)$

where i,j are positive integers and $i \leq j$. By transforming the above equation:

$2n+1=2(i+j+2ij)+1$

we get:

$n=i+j+2ij$

This means that every composite odd number (which is NOT prime) can be generated using this formula. So, by marking numbers of the form $i+j+2ij$, we eliminate non-prime odd numbers.

SIEVE OF SUNDARAM USING BACKTRACKING

Step-by-Step Visualization

$(i+j+2ij\ ,\ 2X+1)$

Example: Find Primes for N = 10

- Step 1: Create an Array
  - Non-prime array (Initially all False):
  - Index: 0 1 2 3 4 5 6 7 8 9 10
  - Value: F F F F F F F F F F F
- Step 2: Mark Non-Primes Using Backtracking
  - Start with i = 1
    - Try j = 1
      - 1 + 1 + 2(1×1) = 4 → Mark index 4
    - Try j = 2
      - 1 + 2 + 2(1×2) = 7 → Mark index 7
    - Try j = 3
      - 1 + 3 + 2(1×3) = 10 → Mark index 10

Backtrack and Try i = 2

- Try j = 2
  - 2 + 2 + 2(2×2) = 10 → Already marked
- Try j = 3 (Exceeds N, so backtrack again)

Final Array After Backtracking:

Index: 0 1 2 3 4 5 6 7 8 9 10

Value: F F F F T F F T F F T

Step 3: Convert Remaining Numbers

Remaining False values:

Index: 0 1 2 3 - 5 6 - 8 9 -

Convert them using 2X + 1:

(2×1)+1 = 3

(2×2)+1 = 5

(2×3)+1 = 7

(2×5)+1 = 11

(2×6)+1 = 13

Final Prime Numbers: {3, 5, 7, 11, 13}

PERMUTATION AND COMBINATION USING BACKTRACKING

Problem Statement

Given an array or string, generate all possible permutations or all possible combinations of the elements.

- Permutation: All possible arrangements of the elements.
- Combination: Select k elements without considering order.

Backtracking Approach

1. Permutations:
    - Swap elements to generate a new permutation.
    - Recursively generate permutations for the remaining elements.
    - Swap back (backtrack) to restore the original order.
2. Combinations:
    - Select an element and include it.
    - Recursively select remaining elements.
    - Backtrack to explore other possibilities.

PERMUTATION AND COMBINATION USING BACKTRACKING

Step-by-Step Explanation of Permutations Using Backtracking

Let’s generate all permutations of "ABC" step by step using backtracking.

Problem Statement

Given a string "ABC", generate all possible permutations (order matters).

Backtracking Process

1. Start with the first character and swap it with itself and others.
2. Recursively permute the rest of the string.
3. Backtrack by undoing the swap to explore other options.

PERMUTATION AND COMBINATION USING BACKTRACKING

Step-by-Step Visualization

Initial String: "ABC"

Fix 'A' at position 0 and permute "BC"

1. [A]BC
   a. Swap 'B' with itself → "ABC"
   b. Swap 'B' and 'C' → "ACB"

Fix 'B' at position 1 and permute "C"

1. A[B]C → "ABC"
2. A[C]B → "ACB"
   a. Generated Permutations: "ABC", "ACB"

Backtrack to 'A' and Swap with 'B'

- Swap 'A' and 'B' → "BAC"

[B]AC

Fix 'A' at position 1 and permute "C"

B[A]C → "BAC"

B[C]A → "BCA"

Generated Permutations: "BAC", "BCA"

Backtrack to 'A' and Swap with 'C'

- Swap 'A' and 'C' → "CBA"

[C]BA

Fix 'B' at position 1 and permute "A"

C[B]A → "CBA"

C[A]B → "CAB"

Generated Permutations: "CBA", "CAB"

Final Output

🔹 All permutations of "ABC" are:

ABC

ACB

BAC

BCA

CBA

CAB

PERMUTATION AND COMBINATION USING BACKTRACKING

Backtracking Tree Diagram

```
           ABC
         /  |  \
      ABC  ACB  BAC
     /   \     /   \
  ABC    ACB  BAC    BCA
        /   \       /   \
     CBA    CAB  CBA    CAB
```

KNIGHT’S TOUR PROBLEM USING BACKTRACKING

Problem Statement

The Knight’s Tour Problem is a classic combinatorial problem in which a knight must visit every square of an N × N chessboard exactly once, following standard knight moves.

Given an N × N chessboard, place a knight on any starting position (x, y) and determine a sequence of moves such that the knight covers every cell without revisiting any cell. If such a sequence exists, print the path; otherwise, return "No solution found."

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

Understanding the Knight's Moves

A knight in chess moves in an "L" shape. It can jump:

- Two squares in one direction, then one square perpendicular
- One square in one direction, then two squares perpendicular

This results in eight possible moves for any position (x, y):

(-2, -1)   (-1, -2)   (+1, -2)   (+2, -1)

(-2, +1)   (-1, +2)   (+1, +2)   (+2, +1)

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

Algorithm (Backtracking Approach)

Since there are multiple paths the knight can take, we use backtracking to explore all possible moves recursively.

Step-by-Step Explanation

1️⃣ Start at the given position (0,0)

2️⃣ Mark it as visited (assign move number 0).

3️⃣ Try all 8 possible knight moves in a loop.

- If the move leads to an unvisited cell, recurse.
- If a move does not lead to a solution, backtrack and try another.
- 4️⃣ If all cells are visited, print the solution.
- 5️⃣ If no valid moves are left, return "No solution found".

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

	0	1	2	3	4
0	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1

THE KNIGHT STARTS AT (0,0) (MOVE 0), MARKING THAT CELL.

STEP 1: KNIGHT MOVES TO (2, 1)

	0	1	2	3	4
0	0	-1	-1	-1	-1
1	-1	1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1

MOVE 1: THE KNIGHT MOVES TO POSITION (2, 1).

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

	0	1	2	3	4
0	0	11	8		
1		14	1	6	9
2	12	7	10	3	
3		2	13		5
4			4		

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

	0	1	2	3	4
0					
1			13	2	9
2	12	1	8	5	
3		6	3	10	
4		11		7	4

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

STEP 2: KNIGHT MOVES TO (4, 2)

	0	1	2	3	4
0	0	-1	-1	-1	-1
1	-1	1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1
4	-1	-1	2	-1	-1

MOVE 2: THE KNIGHT MOVES TO (4, 2).

STEP 3: KNIGHT MOVES TO (3, 4)

	0	1	2	3	4
0	0	-1	-1	-1	-1
1	-1	1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	3
4	-1	-1	2	-1	-1

MOVE 3: THE KNIGHT MOVES TO (3, 4).

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

STEP 4: KNIGHT MOVES TO (1, 3)

	0	1	2	3	4
0	0	-1	-1	4	-1
1	-1	1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	3
4	-1	-1	2	-1	-1

MOVE 4: THE KNIGHT MOVES TO (1, 3).

STEP 5: KNIGHT MOVES TO (2, 5)

	0	1	2	3	4
0	0	-1	-1	-1	5
1	-1	1	-1	-1	-1
2	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	3
4	-1	-1	2	-1	-1

MOVE 5: THE KNIGHT MOVES TO (2, 5).

KNIGHT'S TOUR PROBLEM USING BACKTRACKING

5	22	17	12	7
16	11	6	23	18
21	4	25	8	13
10	15	2	19	24
3	20	9	14	1

SUDOKU SOLVER USING BACKTRACKING

Problem Statement

Sudoku is a 9×9 grid-based puzzle where some cells are pre-filled with numbers (1-9), and the goal is to fill the remaining cells while following these rules:

1. Each row must contain digits 1-9, with no repetition.
2. Each column must contain digits 1-9, with no repetition.
3. Each 3×3 sub-grid must contain digits 1-9, with no repetition.

SUDOKU SOLVER USING BACKTRACKING

```
5  3  _  |  _  7  _  |  _  _  _
6  _  _  |  1  9  5  |  _  _  _
_  9  8  |  _  _  _  |  _  6  _
- - - - - - - - - - - - - - - - - - - - -
8  _  _  |  _  6  _  |  _  _  3
4  _  _  |  8  _  3  |  _  _  1
7  _  _  |  _  2  _  |  _  _  6
- - - - - - - - - - - - - - - - - - - - -
_  6  _  |  _  _  _  |  2  8  _
_  _  _  |  4  1  9  |  _  _  5
_  _  _  |  _  8  _  |  _  7  9
```

SUDOKU SOLVER USING BACKTRACKING

Algorithm (Using Backtracking)

Step 1: Find an Empty Cell

- Traverse the grid from (0,0) to (8,8).
- If an empty cell (_) is found, try filling it with numbers 1-9.

Step 2: Check for Validity

For each number placed, check:

1. If it's not already present in the row.
2. If it's not already present in the column.

If it's not already present in the corresponding 3×3 sub-grid.

Step 3: Place the Number and Recur

- If the number is valid, place it and recursively solve for the next empty cell.
- If it leads to a solution, return true.

Step 4: Backtrack (If Needed)

- If no number leads to a valid solution, undo the last placement (set back to _) and try another number.
- If all numbers fail, backtrack to the previous empty cell.

SUDOKU SOLVER USING BACKTRACKING

Initial Unsolved Sudoku Grid

The empty cells are represented as _.

5	3	_	_	7	_	_	_	_
6	_	_	1	9	5	_	_	_
_	9	8	_	_	_	_	6	_
8	_	_	_	6	_	_	_	3
4	_	_	8	_	3	_	_	1
7	_	_	_	2	_	_	_	6
_	6	_	_	_	_	2	8	_
_	_	_	4	1	9	_	_	5
_	_	_	_	8	_	_	7	9

Step-by-Step Solution with Backtracking

We start by filling empty cells, following Sudoku constraints.

Step 1: Fill (0,2)

- Try 1: ❌ (Already in row)
- Try 2: ❌ (Already in row)
- Try 4: ✅ (Valid, place 4)
-

Step 2: Fill (0,3)

- Try 1: ✅ (Valid, place 1)

SUDOKU SOLVER USING BACKTRACKING

Backtracking Demonstration

If at any step a number leads to a dead-end (no valid moves), we backtrack:

1. Remove last placed number.
2. Try next possible number in the previous cell.
3. Repeat until a valid configuration is found.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

M-COLORING PROBLEM USING BACKTRACKING

Problem Statement

Given an undirected graph with V vertices and E edges, the task is to color the graph using at most M colors, such that no two adjacent vertices share the same color.

If a solution exists, return the coloring of the vertices. Otherwise, report that coloring is not possible..

Adjacency list representation:

- 1 → {2, 3}
- 2 → {1, 4}
- 3 → {1, 4}
- 4 → {2, 3}

Let’s solve it with M = 3 colors {Red, Green, Blue}.

M-COLORING PROBLEM USING BACKTRACKING

Step-by-Step Backtracking Solution

Step 1: Start with Vertex 1

- Try Red ✅ Valid
- Assign Color 1 (Red)

Vertex	Color
1	Red

Step 2: Move to Vertex 2

- Try Red ❌ Invalid (conflict with vertex 1)
- Try Green ✅ Valid
- Assign Color 2 (Green)

Vertex	Color
1	Red
2	Green

M-COLORING PROBLEM USING BACKTRACKING

Step 3: Move to Vertex 3

- Try Red ❌ Invalid (conflict with vertex 1)
- Try Green ✅ Valid
- Assign Color 2 (Green)

Vertex	Color
1	Red
2	Green
3	Green

Step 4: Move to Vertex 4

- Try Red ✅ Valid
- Assign Color 1 (Red)

Vertex	Color
1	Red
2	Green
3	Green
4	Red

M-COLORING PROBLEM USING BACKTRACKING

Summary

✅ Step 1: Start with vertex 1

✅ Step 2: Try each color, ensuring adjacent vertices don't have the same color

✅ Step 3: If valid, move to the next vertex

✅ Step 4: If stuck, backtrack and try another color

✅ Step 5: Continue until all vertices are colored

Backtracking Case (If Coloring Fails)

If at some point, we cannot assign a valid color to a vertex, we backtrack:

1. Remove the last assigned color.
2. Try the next color.
3. If no colors are available, backtrack further.

M-COLORING PROBLEM USING BACKTRACKING

Graph Representation

```
   (1) -- (2)
    |      |
   (3) -- (4)
```

Let’s consider a case where M = 2 (only Red & Green).

M-COLORING PROBLEM USING BACKTRACKING

Step	Vertex	Attempted Color	Is Valid?	Action
1	1	Red	✅ Yes	Assign Red
2	2	Red	❌ No (conflict with 1)	Try next color
3	2	Green	✅ Yes	Assign Green
4	3	Red	❌ No (conflict with 1)	Try next color
5	3	Green	✅ Yes	Assign Green
6	4	Red	❌ No (conflict with 3)	Try next color

M-COLORING PROBLEM USING BACKTRACKING

Step	Vertex	Attempted Color	Is Valid?	Action
7	4	Green	❌ No (conflict with 2)	Backtrack!
8	3	Green Removed	⬅ Backtrack	Try next color
9	3	No valid color	❌ No	Backtrack further!
10	2	Green Removed	⬅ Backtrack	Try next color
11	2	No valid color	❌ No	Backtrack further!
12	1	Red Removed	⬅ Backtrack	Try next color

M-COLORING PROBLEM USING BACKTRACKING

Step	Vertex	Attempted Color	Is Valid?	Action
13	1	Green	✅ Yes	Assign Green
14	2	Red	✅ Yes	Assign Red
15	3	Green	✅ Yes	Assign Green
16	4	Red	✅ Yes	Assign Red

✅ Solution Found: {1 → Green, 2 → Red, 3 → Green, 4 → Red}

Key Observations

- If we get stuck, we remove the last assigned color (backtrack) and try a new one.
- If no colors work, we go back even further and change the previous vertex's color.
- If we find a valid path, we proceed forward again.

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Problem Statement:

Given a graph represented as an adjacency matrix, determine if there exists a Hamiltonian cycle. A Hamiltonian cycle is a path in an undirected graph that visits every vertex exactly once and returns to the starting vertex.

Algorithm Explanation:

We use backtracking to explore all possible paths, ensuring that each vertex is visited only once before returning to the starting vertex.

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Step-by-Step Backtracking Algorithm:

1. Initialize Path:
    - Create an array path[] to store the cycle.
    - Set path[0] to the starting vertex (typically vertex 0).
2. Recursive Function to Build Path (hamiltonianUtil):
    - Try adding the next vertex (v) to the path.
    - Check constraints:
        - v must be adjacent to the last vertex in path[].
        - v should not already be in path[] (no revisits).
    - If valid, add v to path[] and recursively call hamiltonianUtil for the next position.
    - If adding v leads to a solution, return true.
    - If not, backtrack: remove v from path[] and try another vertex.

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

- Base Case (Solution Found):
  - If all vertices are in path[] and the last vertex connects to the starting vertex, return true.
- Backtracking if No Solution:
  - If no valid vertex is found, return false.

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Example Graph Representation

Consider the following graph:

```
(0) --- (1)
 |\      |
 | \     |
 |  \    |
(3) --- (2)
```

Edges:

- (0 → 1), (0 → 2), (0 → 3), (1 → 2), (2 → 3), (3 → 0)

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Step-by-Step Backtracking Process

- Graph as an Adjacency Matrix

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Step-by-Step Hamiltonian Path Construction

Step	Path	Action
1	[0]	Start from vertex 0
2	[0, 1]	Add vertex 1 (valid)
3	[0, 1, 2]	Add vertex 2 (valid)
4	[0, 1, 2, 3]	Add vertex 3 (valid)
5	[0, 1, 2, 3, 0]	Complete the cycle!

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Backtracking Approach

1. Start from Vertex 0 and initialize the path as [0].
2. Try adding an adjacent vertex that has not been visited.
3. If no valid move exists, backtrack and try another path.
4. If all vertices are in the path and the last vertex connects back to the first vertex, we found a Hamiltonian Cycle.

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

What if We Chose the Wrong Path? (Backtracking Example)

If we made a wrong choice, we would backtrack.

Step	Path (Current)	Action Taken	Backtracking?
1	[0]	Start from vertex 0	No
2	[0, 3]	Add vertex 3 (valid)	No
3	[0, 3, 2]	Add vertex 2 (valid)	No
4	[0, 3, 2, 1]	Add vertex 1 (valid)	No
5	[0, 3, 2, 1]	No edge from 1 to 0 → Backtrack to 2	Yes

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

What if We Chose the Wrong Path? (Backtracking Example)

If we made a wrong choice, we would backtrack.

Path (Current)	Action Taken	Backtracking?
[0, 3]	Remove 2, try another path	Yes
[0, 3, 1]	Add vertex 1 (valid)	No
[0, 3, 1, 2]	Add vertex 2 (valid)	No
[0, 3, 1, 2, 0]	Cycle completed!	No
[0, 3, 2, 1]	No edge from 1 to 0 → Backtrack to 2	Yes

New Hamiltonian Cycle Found:

0 → 3 → 1 → 2 → 0

HAMILTONIAN CYCLE PROBLEM USING BACKTRACKING

Key Observations

1. The algorithm explores all possible paths.
2. If an invalid path is encountered, it backtracks and retries a different vertex.
3. The final solution is a valid Hamiltonian Cycle.

DYNAMIC PROGRAMMING

What is Dynamic Programming?

Dynamic Programming is an optimization technique used for solving problems by breaking them down into smaller subproblems, solving each only once, and storing their results to avoid recomputing them.

It is particularly useful for problems that exhibit:

1. Optimal Substructure → The solution to the problem can be built from solutions to its subproblems.
2. Overlapping Subproblems → The same subproblem is solved multiple times.

DYNAMIC PROGRAMMING

Key Features of Dynamic Programming

Feature	Description
Overlapping Subproblems	A problem is broken down into smaller subproblems that are solved repeatedly.
Optimal Substructure	The solution to a larger problem can be constructed from solutions to its smaller subproblems.
Memoization (Top-Down)	Storing results of expensive function calls to reuse them later.
Tabulation (Bottom-Up)	Solving all subproblems iteratively and building solutions in a table.

DYNAMIC PROGRAMMING

Types of Dynamic Programming Approaches

1. Top-Down Approach (Memoization)
    - Uses recursion and caching.
    - Stores solutions to subproblems in an array (or hash table) to avoid redundant computations.
2. Bottom-Up Approach (Tabulation)
    - Fills a table iteratively instead of using recursion.
    - Works best when all subproblems are needed.

DYNAMIC PROGRAMMING

Steps for Solving Problems using Dynamic Programming

1. Identify if DP is applicable
    - Check for overlapping subproblems and optimal substructure.
2. Define the State (Subproblems)
    - What does dp[i] represent?
3. Formulate the Recurrence Relation
    - Define how dp[i] depends on previous states.
4. Choose between Memoization or Tabulation
    - If recursion causes redundancy → use memoization.
    - If iterative computation is faster → use tabulation.
5. Compute the Final Answer
    - Use the stored results to derive the final answer efficiently.

GREEDY VS. DYNAMIC PROGRAMMING

Feature	Greedy Algorithm	Dynamic Programming
Approach	Makes locally optimal choices at each step	Solves subproblems and combines solutions optimally
Optimality	May not always guarantee an optimal solution	Always guarantees an optimal solution if applied correctly
Subproblem Dependency	Decisions do not depend on previously solved subproblems	Uses previously solved subproblems (overlapping subproblems property)
Computation al Complexity	Faster (O(n log n) or O(n))	Usually slower (O(n^2) or O(2^n) in worst cases)

GREEDY VS. DYNAMIC PROGRAMMING

Feature	Greedy Algorithm	Dynamic Programming
Space Complexity	Uses less memory	Requires extra space for storing subproblem results
Use Cases	Best for problems with the greedy-choice property and optimal substructure	Best for problems with overlapping subproblems and optimal substructure
Example Problems	Kruskal’s Algorithm, Prim’s Algorithm, Huffman Coding, Fractional Knapsack	Fibonacci Sequence, 0/1 Knapsack, Longest Common Subsequence, Matrix Chain Multiplication

TOP-DOWN VS. BOTTOM-UP APPROACH IN DYNAMIC PROGRAMMING

Dynamic Programming (DP) can be implemented using two main approaches:

1️⃣ Top-Down Approach (Memoization)

2️⃣ Bottom-Up Approach (Tabulation)

Each has its advantages and is used depending on the problem constraints.

TOP-DOWN VS. BOTTOM-UP APPROACH IN DYNAMIC PROGRAMMING

1. Top-Down Approach (Memoization)

Concept

- Solves the problem recursively.
- Stores the results of already computed subproblems to avoid recomputation.
- Uses a cache (array, hashmap, or dictionary) to store results.

Steps

1. Define a recursive function.
2. Check if the result is already computed (in cache).
    - If yes, return the stored value.
    - If no, compute it recursively, store the result, and return it.
3. Recursion continues until the base case is reached.

TOP-DOWN VS. BOTTOM-UP APPROACH IN DYNAMIC PROGRAMMING

2. Bottom-Up Approach (Tabulation)

Concept

- Starts from the smallest subproblems and builds up the final solution iteratively.
- Uses a table to store computed values and avoids recursion.
- Eliminates function call overhead (more efficient).

Steps

1. Identify the base case.
2. Iterate over subproblems and compute each in order.
3. Use previously computed values to solve the current subproblem.
4. Build the solution iteratively.

GREEDY VS. DYNAMIC PROGRAMMING

Feature	Top-Down (Memoization)	Bottom-Up (Tabulation)
Approach	Recursion + Caching	Iterative DP Table
Storage	Uses an array for caching	Uses an array or two variables
Time Complexity	O(n)	O(n)
Space Complexity	O(n) (Recursion stack)	O(n) (Can be reduced to O(1))
Function Calls	Uses recursive calls	Uses loops (no recursion)
Best for?	Problems where only needed subproblems are computed	Problems that require computing all subproblems

TOP-DOWN VS. BOTTOM-UP APPROACH IN DYNAMIC PROGRAMMING

Which One to Use?

✔ Use Top-Down (Memoization) if:

- You only need to solve some subproblems.
- Recursion is easier to implement.

✔ Use Bottom-Up (Tabulation) if:

- All subproblems need to be solved.
- You want better space efficiency (O(1) possible).

SUMMARY

- Both approaches eliminate redundant calculations in recursion.
- Bottom-Up is generally more efficient in space.
- Top-Down is easier to write for many recursive problems.

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Problem Statement

You are given N items, each with a weight and value. You have a knapsack that can carry a maximum weight W.

You need to determine the maximum total value you can obtain by selecting some of the items such that the total weight does not exceed W.

- Condition: You can either pick an item or leave it (0/1 choice).
- Goal: Maximize sum of values without exceeding total weight.

Example

Input

N = 3, W = 50

Items: (value, weight)

(60, 10), (100, 20), (120, 30)

Output

Maximum value = 220

Items selected: (100, 20) + (120, 30)

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Approach: Dynamic Programming (DP)

Since there are overlapping subproblems, we can use DP instead of recursion to optimize it.

Steps to Solve

1. Create a DP table dp[i][w] where:
    - i represents items (1 to N).
    - w represents remaining weight (0 to W).
    - dp[i][w] stores the maximum value that can be obtained using the first i items and weight w.

2. Recurrence Relation:
    - If we exclude item i:
    - dp[i][w] = dp[i-1][w] (value without taking item i).
    - If we include item i (only if w[i] ≤ w):
    - dp[i][w] = max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]]).
    - Take the maximum of the two cases.

3. Fill the table iteratively using the above recurrence.

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Given Problem

We have N = 3 items with the following weights and values:

Item	Value (V)	Weight (W)
1	60	10
2	100	20
3	120	30

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Step 1: Define the DP Table

- Create a 2D DP table dp[i][w] where:
    - i represents the first i items being considered.
    - w represents the current knapsack weight limit.
    - dp[i][w] stores the maximum value possible with i items and weight w.
- Initialize:
    - dp[0][w] = 0 for all w (because with 0 items, the value is 0).
    - dp[i][0] = 0 for all i (because if knapsack weight is 0, we can't add anything).

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Step 2: Build the DP Table

- We iterate over each item.
- For each capacity w, we decide whether to include or exclude the item.

Step 3: Fill the DP Table (Row by Row)

We will fill the DP table for each item and weight limit, following the recurrence formula:

$$dp[i][w]=\max(dp[i-1][w],value[i-1]+dp[i-1][w-weight[i-1]])$$

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Row 2: Considering Items {1,2} (100, 20)

- If $w < 20$, we can't include item 2.
  - We take the previous row's value (dp[1][w]).
- If $w \geq 20$, we choose max of:
  - Exclude item 2 → dp[1][w]
  - Include item 2 → 100 + dp[1][w - 20]

Explanation:

- For w = 20: max(60, 100) = 100
- For w = 30: max(60, 100 + 60) = 160
- For w = 40, 50: The best combination remains (item 2 + item 1) = 160.

Weight (w) →	0	5	10	15	20	25	30	35	40	45	50
Item 1 (60, 10)	0	0	60	60	60	60	60	60	60	60	60
Item 2 (100, 20)	0	0	60	60	100	100	100	160	160	160	160

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Row 3: Considering Items {1,2,3} (120, 30)

- If w < 30, we can't include item 3.
    - We take the previous row's value (dp[2][w]).
- If w ≥ 30, we choose max of:
    - Exclude item 3 → dp[2][w]
    - Include item 3 → 120 + dp[2][w - 30]

Explanation:

- For w = 30: max(160, 120) = 120
- For w = 40: max(160, 120 + 60) = 180
- For w = 50: max(160, 120 + 100) = 220

✅ (Final Answer)

Weight (w) →	0	5	10	15	20	25	30	35	40	45	50
Item 1 (60, 10)	0	0	60	60	60	60	60	60	60	60	60
Item 2 (100, 20)	0	0	60	60	100	100	100	160	160	160	160
Item 3 (120, 30)	0	0	60	60	100	100	120	160	180	220	220

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

Row 3: Considering Items {1,2,3} (120, 30)

- If w < 30, we can't include item 3.
  - We take the previous row's value (dp[2][w]).
- If w ≥ 30, we choose max of:
  - Exclude item 3 → dp[2][w]
  - Include item 3 → 120 + dp[2][w - 30]

Explanation:

- For w = 30: max(160, 120) = 120
- For w = 40: max(160, 120 + 60) = 180
- For w = 50: max(160, 120 + 100) = 220

✅ (Final Answer)

Weight (w) →	0	5	10	15	20	25	30	35	40	45	50
Item 1 (60, 10)	0	0	60	60	60	60	60	60	60	60	60
Item 2 (100, 20)	0	0	60	60	100	100	100	160	160	160	160
Item 3 (120, 30)	0	0	60	60	100	100	120	160	180	220	220

0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

For N = 3 items and W = 50, we build the DP table.

Items ↓ \ Capacity →	0	10	20	30	40	50
No items (0)	0	0	0	0	0	0
Item 1 (60,10)	0	60	60	60	60	60
Item 2 (100,20)	0	60	100	100	160	160
Item 3 (120,30)	0	60	100	120	160	220

LONGEST COMMON SUBSEQUENCE

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

LONGEST COMMON SUBSEQUENCE

Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

For => S1 = {B, C, D, A, A, C, D}

{A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

LONGEST COMMON SUBSEQUENCE

For,

S1 = {B, C, D, A, A, C, D}

S2 = {A, C, D, B, A, C}

Then, common subsequences are

{B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...

Among these subsequences, **{C, D, A, C}** is the longest common subsequence.

USING DYNAMIC PROGRAMMING TO FIND THE LCS

Two sequences,

X	A	C	A	D	B
Y	C	B	D	A	

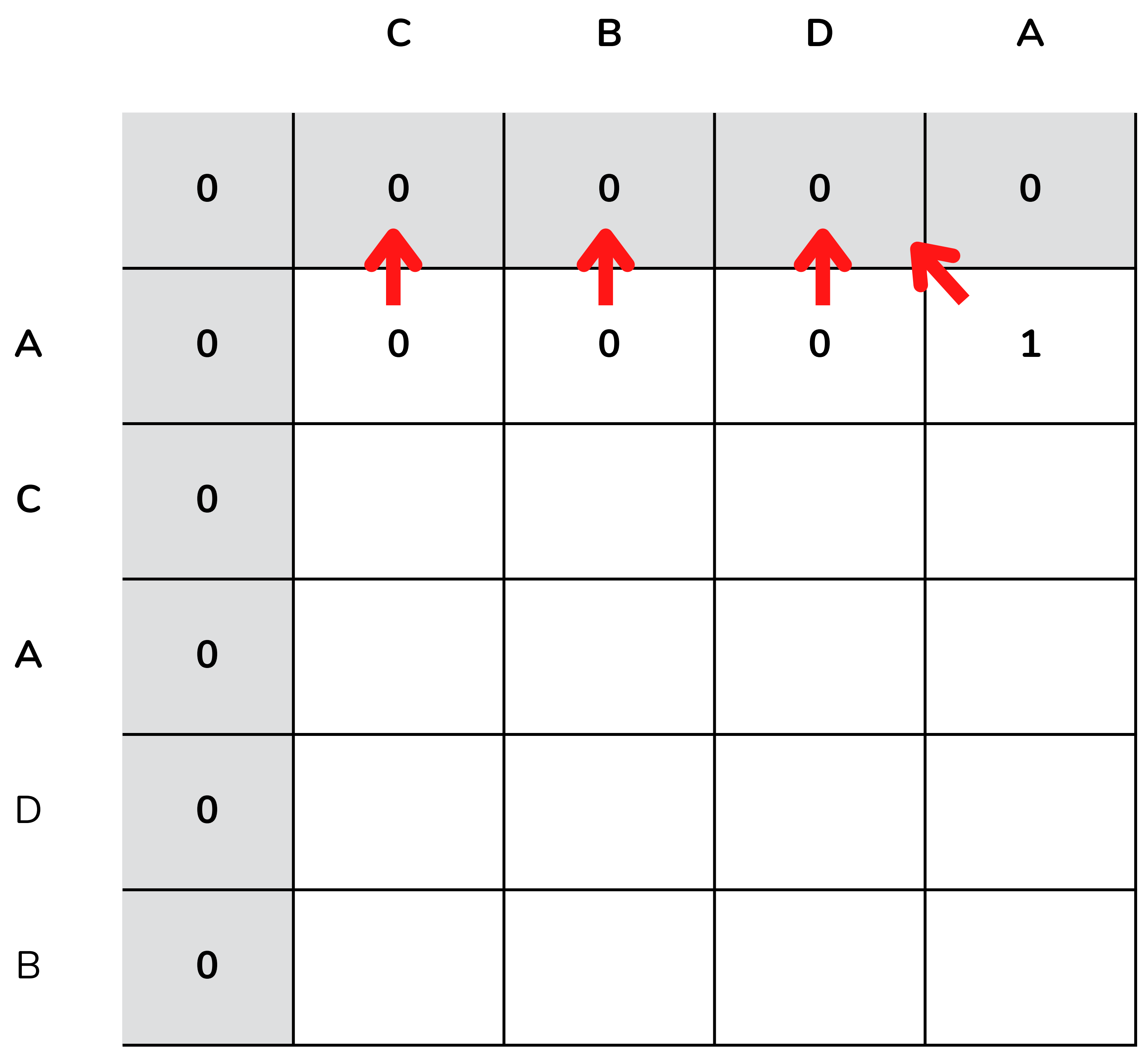
Create a table (matrix) of dimensions, (n+1)*(m+1) where n and m are the lengths of X and Y. The first rows and columns are filled with 0s

USING DYNAMIC PROGRAMMING TO FIND THE LCS

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

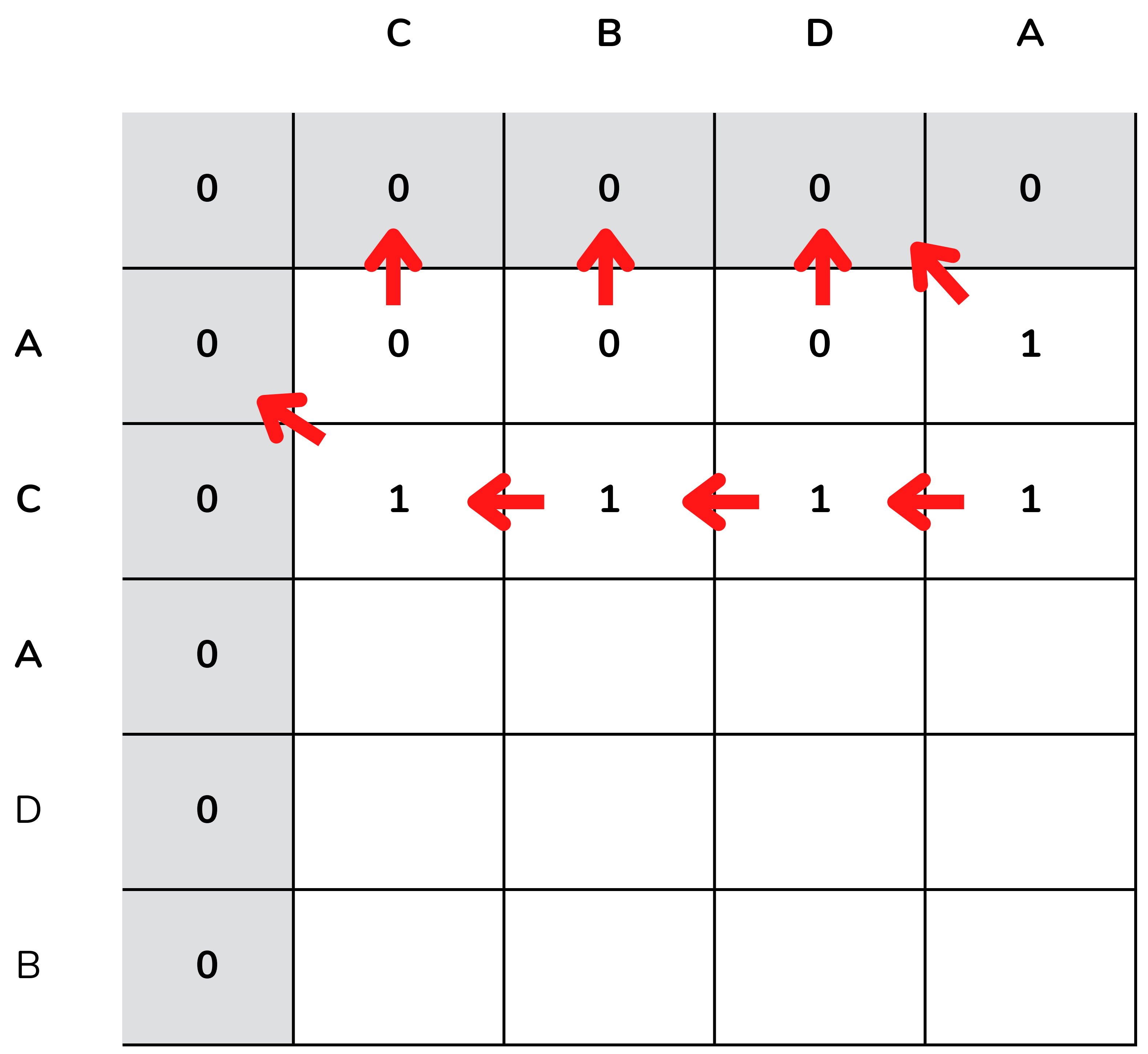
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1
C	0				
A	0				
D	0				
B	0				

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

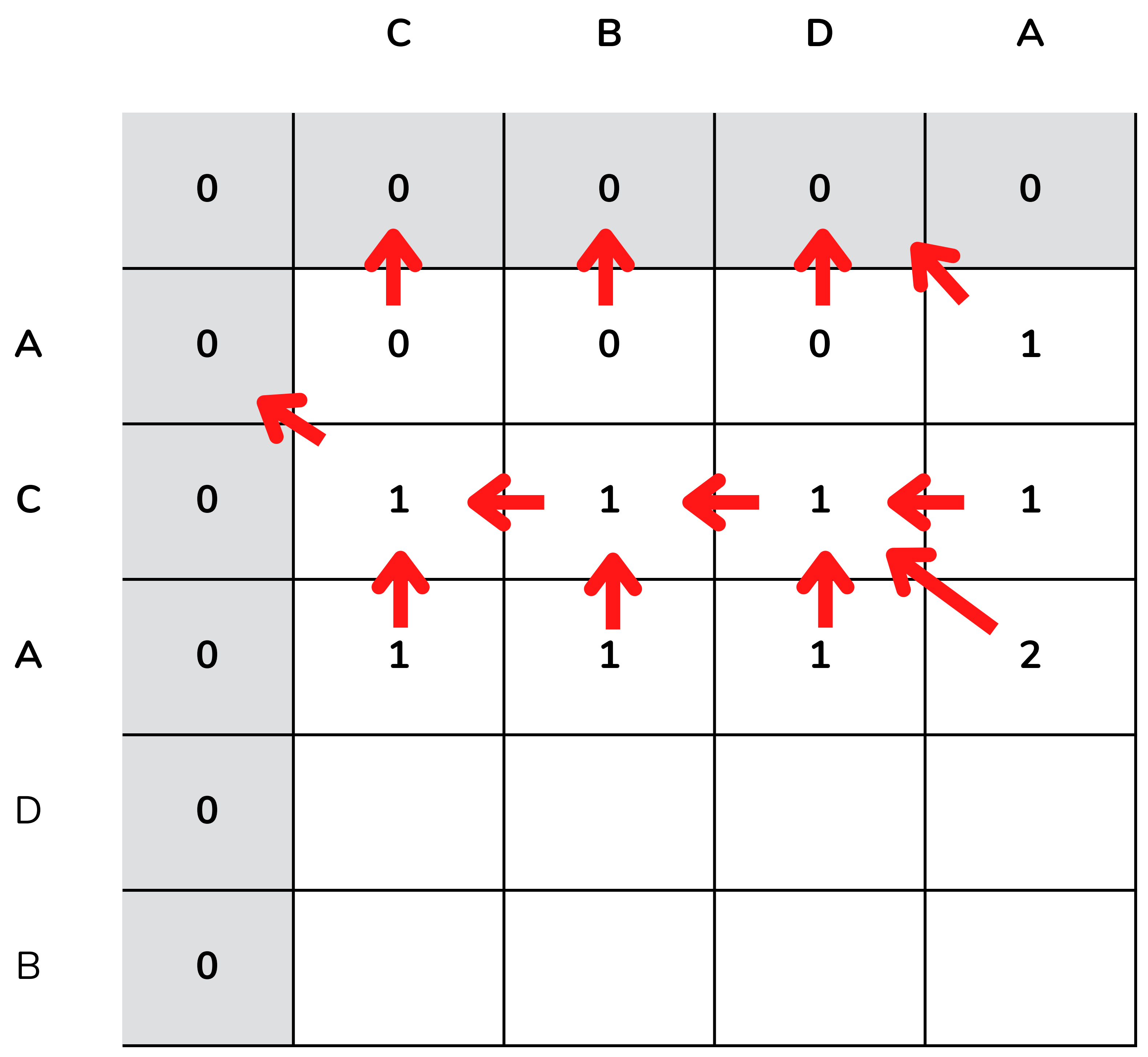
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0				
D	0				
B	0				

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

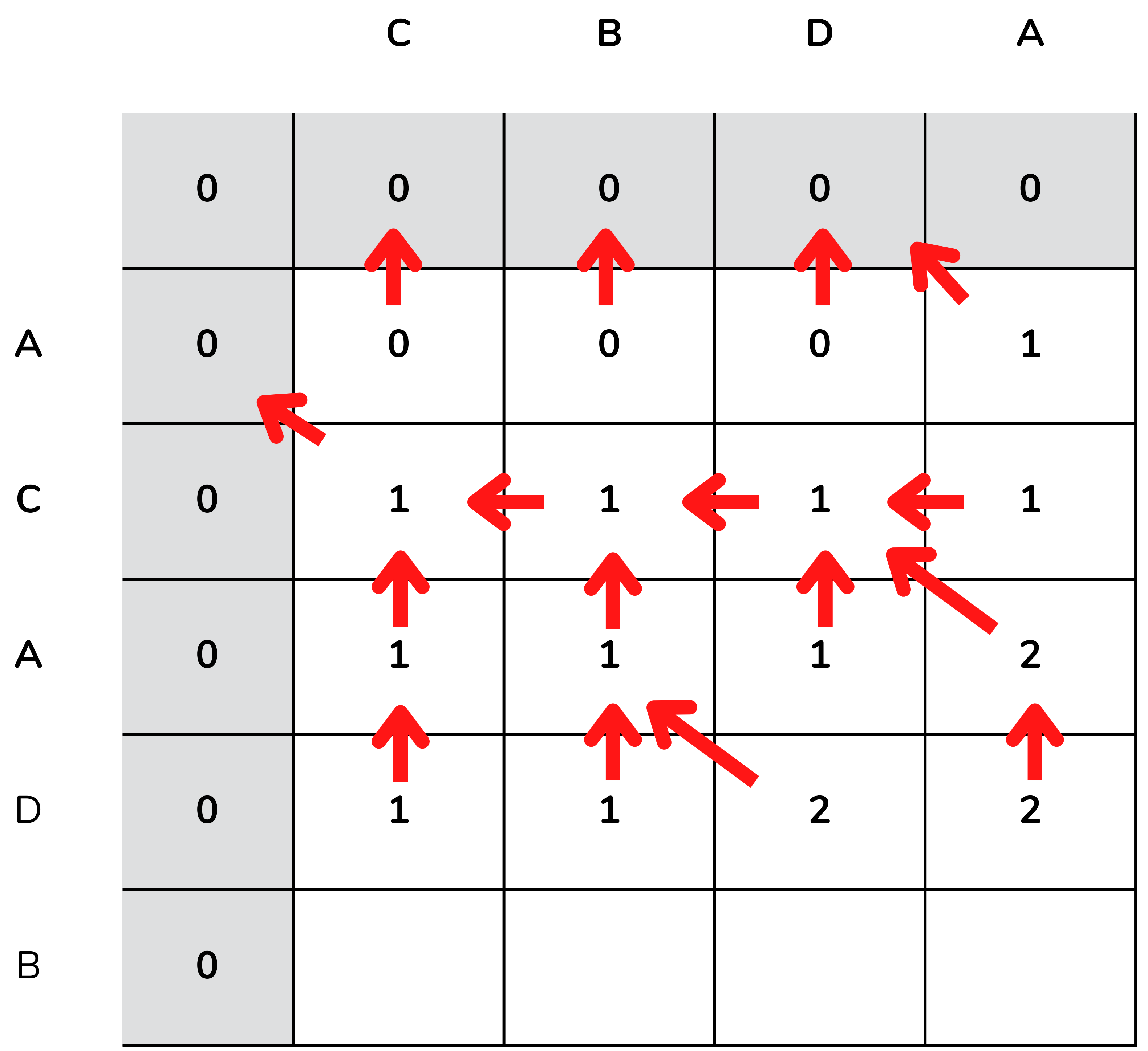
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1
C	0	↖ 1	← 1	← 1	← 1
A	0	↑ 1	↑ 1	↑ 1	↖ 2
D	0				
B	0				

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

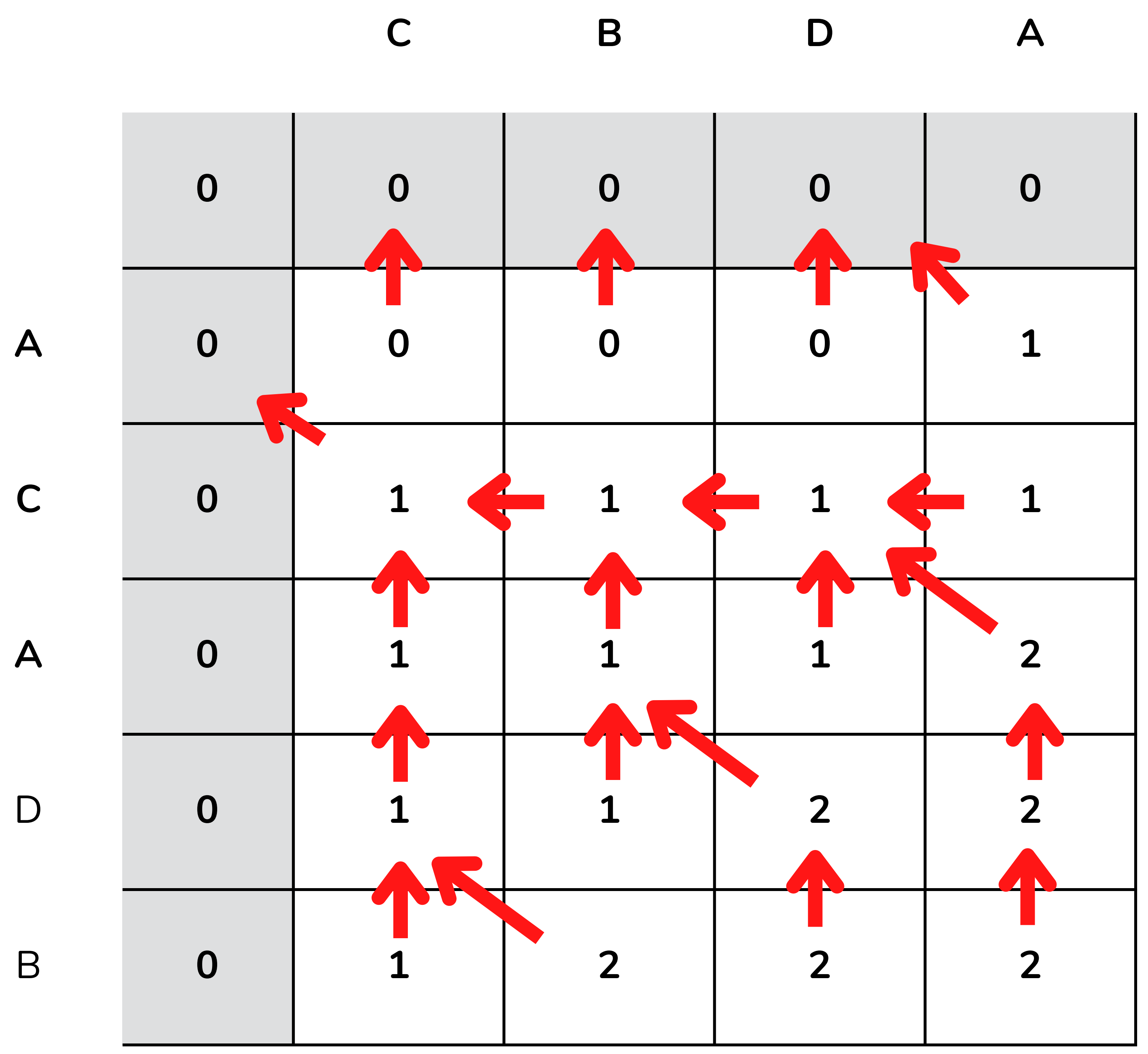
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0				

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

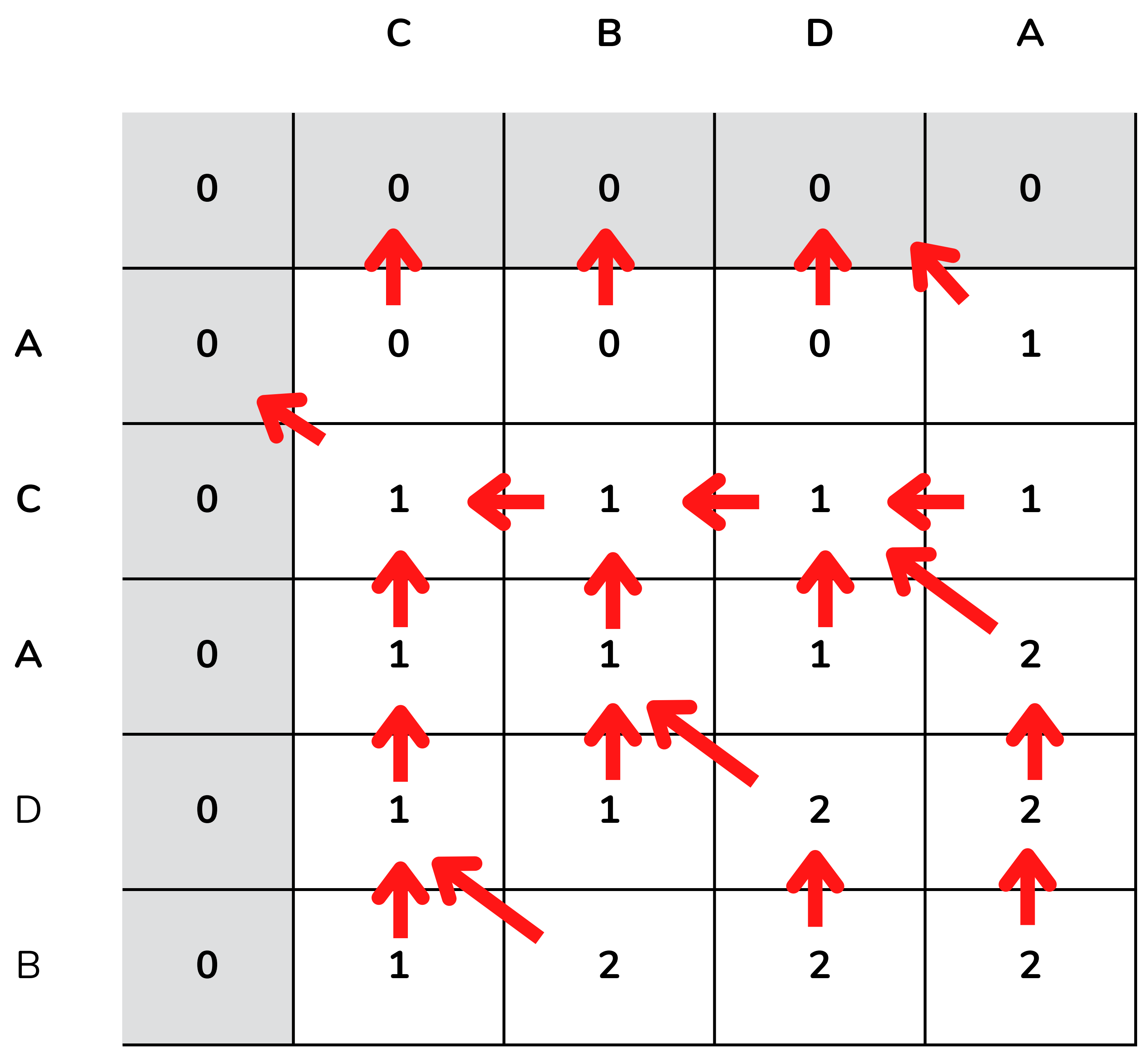
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

1. Fill each cell of the table using the following logic.
2. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
3. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.
4. Fill the values
5. Step 2 is repeated until the table is filled.
6. Fill all the values
7. The value in the last row and the last column is the length of the longest common subsequence.

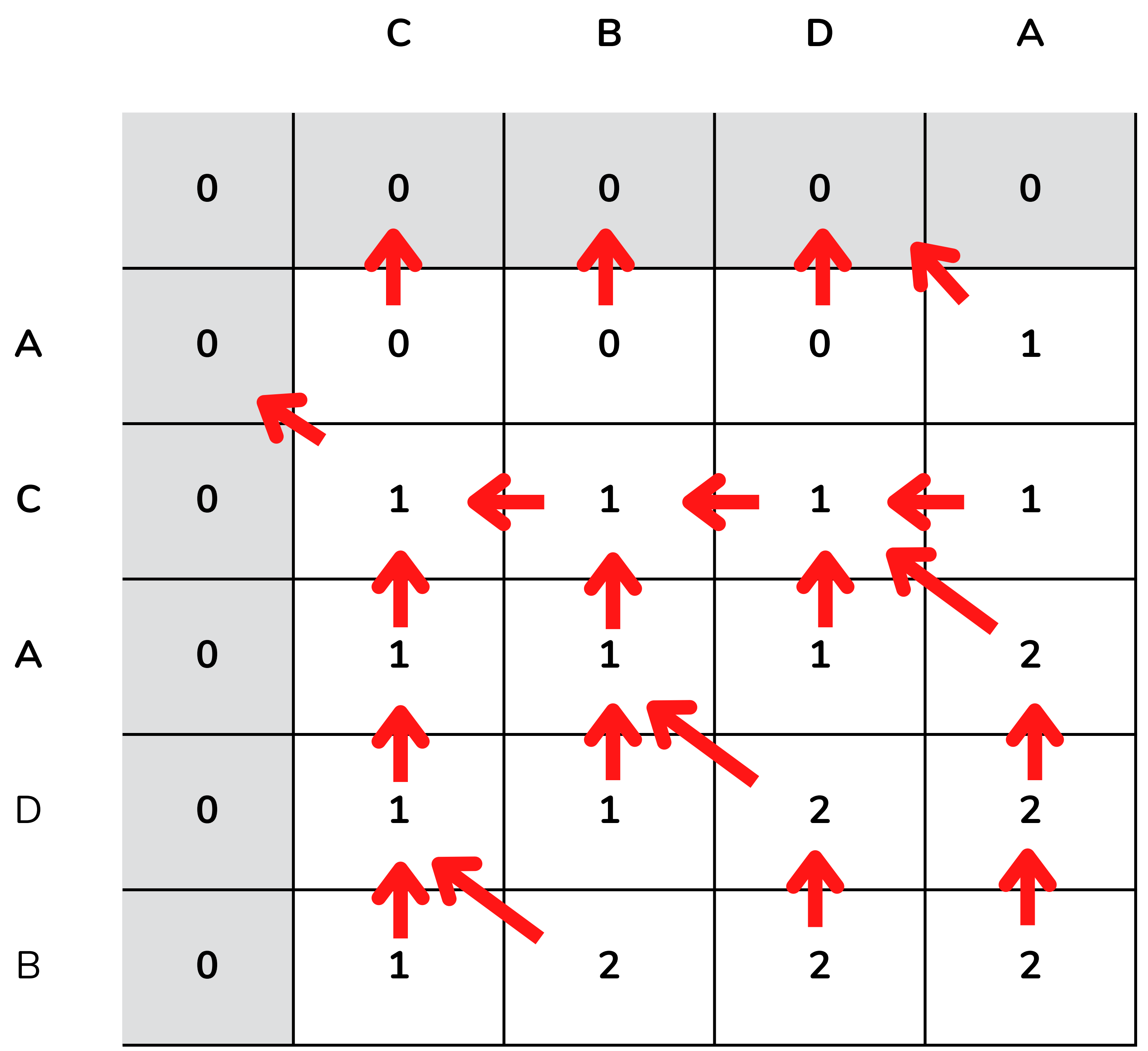
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

The value in the last row and the last column is the length of the longest common subsequence is 2

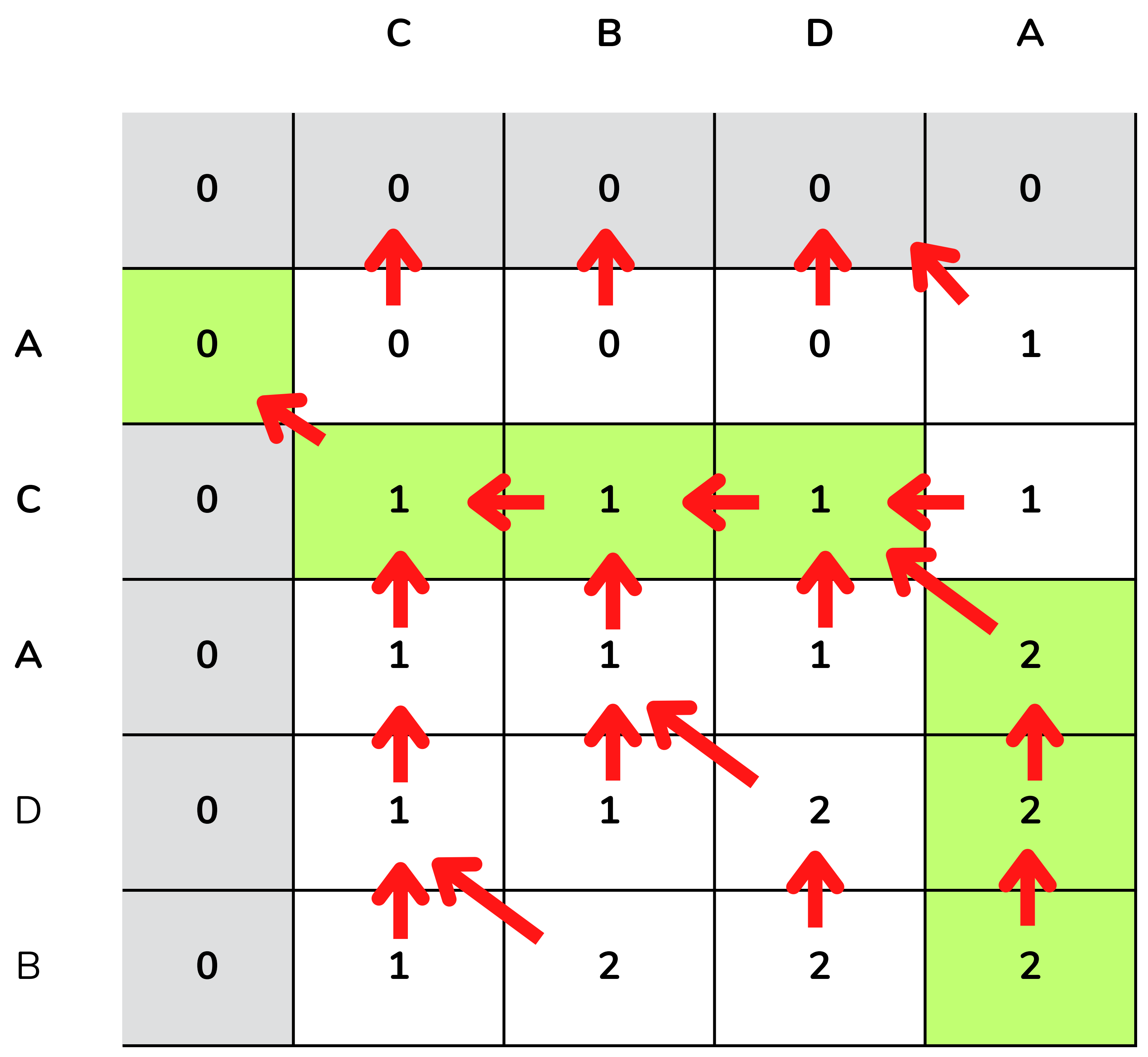
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.

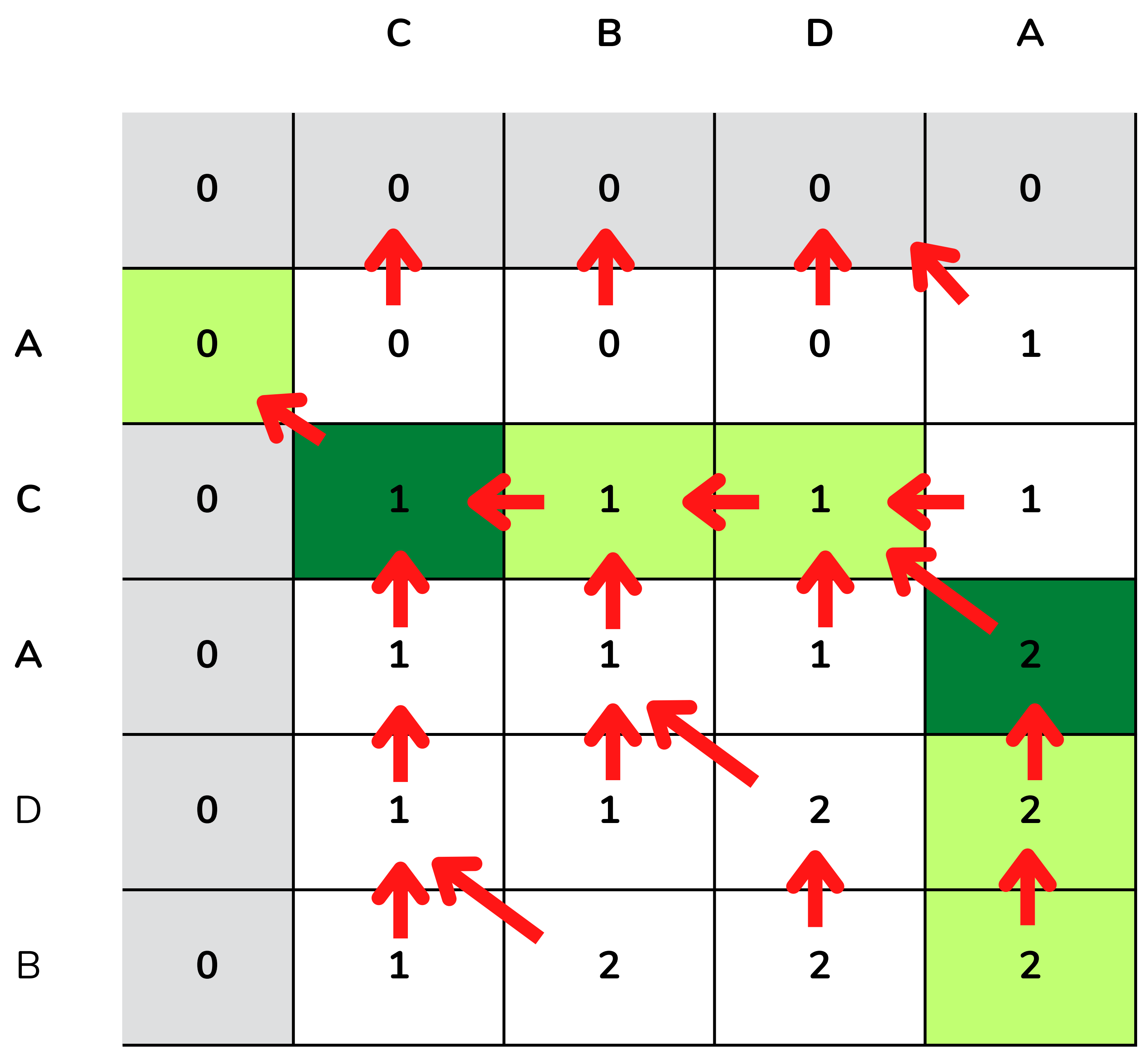
USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

After tracing the path, select the elements with diagonal arrows only.

USING DYNAMIC PROGRAMMING TO FIND THE LCS



		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0	1	1	1	1
A	0	1	1	1	2
D	0	1	1	2	2
B	0	1	2	2	2

After tracing the path, select the elements with diagonal arrows only.

Print the corresponding elements:

CA

USE CASES FOR LCS

Variations of this problem are commonly found in real-world applications such as bioinformatics, natural language processing, and text comparison.

USE CASES FOR LCS

1. Bioinformatics

- DNA Sequence Analysis: LCS is used to identify similarities between DNA sequences. By finding common subsequences in DNA, researchers can determine evolutionary relationships, functional similarities, and genetic mutations.

2. Version Control Systems

- Diff Tools: Version control systems like Git use LCS algorithms to compare different versions of files. By identifying the longest common subsequence, these tools can highlight the differences and merge changes effectively.

USE CASES FOR LCS

3. Text Comparison

- Document Comparison: LCS is used in tools that compare different versions of documents to identify changes, additions, or deletions. This is useful in word processors, plagiarism detection tools, and legal document analysis.

4. Data Compression

- Compression Algorithms: LCS can be used in data compression techniques where common subsequences are identified and encoded efficiently to reduce the overall size of the data.

LONGEST INCREASING SUBSEQUENCE

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.

LONGEST INCREASING SUBSEQUENCE

This subsequence is not necessarily contiguous or unique. The longest increasing subsequences are studied in the context of various disciplines related to mathematics, including algorithmics, random matrix theory, representation theory, and physics.

FINDING THE LONGEST INCREASING SUBSEQUENCE

APPLICAITONS

- Bioinformatics: Finding longest chains in protein sequences.
- Stock Market Analysis: Identifying periods of increasing stock prices.
- Data Analysis: Finding trends in time-series data.

LONGEST INCREASING SUBSEQUENCE

Example:

Consider the array: [10, 9, 2, 5, 3, 7, 101, 18].

The longest increasing subsequences are:

[2, 3, 7, 18]

[2, 5, 7, 101]

Both of these subsequences have a length of 4. Therefore, the length of the longest increasing subsequence is 4.

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

- Create an array dp where dp[i] represents the length of the longest increasing subsequence ending at index i.
- Initialize all values of dp to 1 because the minimum length of LIS ending at any element is 1 (the element itself).

arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i = 1; j = 0;

j < i => True

	j	i						
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

a[i] > a[j] => False; No Update

j++ => j < i => False; i++

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=2; j = 0;

j < i => True

	j		i					
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

a[i] > a[j] => False; No Update

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=2; j = 1;

j < i => True

		j	i					
arr	10	9	2	5	3	7	101	18
dp	1	1	1	1	1	1	1	1

a[i] > a[j] => False; No Update

j++ => j < i => False; i++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=3; j = 0;

j < i => True

	j			i				
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

a[i] > a[j] => False; No Update

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=3; j = 1;
j < i => True

	j		i					
arr	10	9	2	5	3	7	101	18
dp	1	1	1	1	1	1	1	1

a[i] > a[j] => False; No Update
j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=3; j = 2;

j < i => True

			j	i				
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	1	1	1	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => False; i++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=4; j = 0;

j < i => True

	j				i			
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	1	1	1	1

a[i] > a[j] => False;

j++ => j < i => False; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=4; j = 1;

j < i => True

		j			i			
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**1**	**1**	**1**	**1**

a[i] > a[j] => False;

j++ => j < i => False; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=4; j = 2;

j < i => True

			j		i			
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**1**	**1**	**1**

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => False; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i=4; j = 3;

j < i => True

				j	i			
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	1	1	1

a[i] > a[j] => False;

j++ => j < i => False; i++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=5; j = 0;

j < i => True

	j					i		
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	1	1	1

a[i] > a[j] => False;

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=5; j = 1;

j < i => True

		j				i		
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	1	1	1

a[i] > a[j] => False;

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=5; j = 2;
j < i => True

			j			i		
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**2**	**1**	**1**

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);
j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=5; j = 3;

j < i => True

				j		i		
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	1	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=5; j = 4;

j < i => True

					j	i		
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	1	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => False; i++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 0;

j < i => True

	j						i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	2	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 1;

j < i => True

		j					i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	2	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 2;

j < i => True

			j				i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	2	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 3;

j < i => True

				j			i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	3	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 4;

j < i => True

					j		i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	3	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=6; j = 5;

j < i => True

						j	i	
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	4	1

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => False; i++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 0;

j < i => True

	j							i
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	4	2

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 1;

j < i => True

		j						i
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	4	2

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING THE LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 2;

j < i => True

			j					i
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	4	2

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 3;

j < i => True

				j				i
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**3**	**4**	**3**

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 4;

j < i => True

					j			i
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**3**	**4**	**3**

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 5;

j < i => True

						j		i
arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**3**	**4**	**4**

a[i] > a[j] => True; dp[i] =max(dp[i], dp[j] + 1);

j++ => j < i => True; j++;

FINDING LONGEST INCREASING SUBSEQUENCE

USING DYNAMIC PROGRAMMING

i=7; j = 6;

j < i => True

							j	i
arr	10	9	2	5	3	7	101	18
dp	1	1	1	2	2	3	4	4

a[i] > a[j] => False;

j++ => j < i => False; i++;

FINDING LONGEST INCREASING SUBSEQUENCE USING DYNAMIC PROGRAMMING

i = 8 => Array out of bounds, termination.

arr	**10**	**9**	**2**	**5**	**3**	**7**	**101**	**18**
dp	**1**	**1**	**1**	**2**	**2**	**3**	**4**	**4**

Thus, the length of the longest increasing subsequence in the array {10, 9, 2, 5, 3, 7, 101, 18} is 4.

APPLICATIONS OF LONGEST INCREASING SUBSEQUENCE WITH REAL-WORLD SCENARIOS

Bioinformatics:

- DNA Sequence Analysis: In bioinformatics, LIS can be used to identify the longest increasing subsequence of nucleotides (DNA, RNA) or amino acids (proteins). This can help in understanding evolutionary relationships, finding conserved regions, and aligning sequences.
- Gene Prediction: When comparing gene sequences, LIS can be used to find similar gene structures and predict functions of unknown genes based on known ones.

2. Computer Vision:

- Object Recognition: In computer vision, LIS algorithms can be used to recognize objects in images by identifying the longest subsequence of features (edges, colors, shapes) that match a known pattern.
- Motion Tracking: LIS can be applied to track the movement of objects over time by identifying the longest sequence of positions that form a coherent path.

COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

The coin change problem is a well-known dynamic programming (DP) problem that requires us to determine the number of ways to make a certain amount using given denominations of coins.

Problem Statement

Given an array coins[] representing available denominations and an integer amount, find the total number of ways to make up that amount using any number of coins.

Example 1
Input:
coins = {1, 2, 5}
amount = 5

Output:
Ways to make 5: 4

Explanation:

There are 4 ways to make 5:

1. 1 + 1 + 1 + 1 + 1
2. 1 + 1 + 1 + 2
3. 1 + 2 + 2
4. 5

COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

Approach: Dynamic Programming

We use Dynamic Programming (DP) to solve this problem in an optimized way.

1. Define the DP State

Let dp[i] be the number of ways to make amount i using given coins.

2. Recursive Formula

For each coin c, update the DP table as:

$$dp[i] = dp[i] + dp[i-c]$$

where:

- dp[i] represents the number of ways to make sum i.
- dp[i - c] is the number of ways to make i after using coin c.

3. Base Case

- dp[0] = 1 (There is one way to make 0: by using no coins).

COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

		0	1	2	3	4
No Coin	0	0	0	0	0	0
Only Coin 1	1	1	1	1	1	1
Coin 1 and Coin 2	2	1	1	2	2	3
All 1, 2 and 3 Coin	3	1	1	2	3	4

COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

Example 1: Suppose you are given the coins 1 cent, 5 cents, and 10 cents with N = 8 cents, what are the total number of combinations of the coins you can arrange to obtain 8 cents.

Input: N=8

Coins : 1, 5, 10

Output: 2

Explanation:

1 way:

1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8 cents.

2 way:

1 + 1 + 1 + 5 = 8 cents.

COIN CHANGE PROBLEM USING DYNAMIC PROGRAMMING

You are given a sequence of coins of various denominations as part of the coin change problem. For example, consider the following array a collection of coins, with each element representing a different denomination.

{ 2, 3, 5, 10, 20, 30, 50 }

For example, if you want to reach 78 using the above denominations, you will need the four coins listed below.

{ 3, 5, 20, 50 }

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Problem Statement

Given a set of N positive integers and a target sum S, determine whether there exists a subset whose sum equals S.

Example:

Input: set = {3, 34, 4, 12, 5, 2}, sum = 9

Output: Yes (Subset {4, 5} forms the sum 9)

Dynamic Programming Approach

We use a 2D DP table where:

- dp[i][j] represents whether a subset of the first i elements has a sum of j.
- Transition formula:
    - If arr[i-1] > j: Exclude the current element → dp[i][j] = dp[i-1][j]
    - If arr[i-1] ≤ j: Include or Exclude the current element →
    - dp[i][j] = dp[i-1][j] OR dp[i-1][j - arr[i-1]]

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Let’s consider an example:

Set: {3, 34, 4, 12, 5, 2}

Target Sum: 9

Step 1: Initialize DP Table

- dp[i][0] = true (A subset sum of 0 is always possible using an empty subset)
- dp[0][j] = false for j > 0 (No subset possible with 0 elements)

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Let’s consider an example:

Set: {3, 34, 4, 12, 5, 2}

Target Sum: 9

Step 1: Initialize DP Table

- dp[i][0] = true (A subset sum of 0 is always possible using an empty subset)
- dp[0][j] = false for j > 0 (No subset possible with 0 elements)

Elements → Sum ↓	∅ (0 elements)
Sum = 0	✅ (Always True)
Sum = 1, 2, 3, ..., 9	❌ (Impossible with 0 elements)

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Filling the Table

We process each element one by one: {3, 34, 4, 12, 5, 2}

For each sum, we check:

1. Excluding current element → Copy value from above row.
2. Including current element → If dp[i-1][j - arr[i-1]] was true, set dp[i][j] = true.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Understanding the DP Table

We define a DP table dp[i][j], where:

- dp[i][j] is true if we can form sum j using the first i elements.
- dp[i][j] is false otherwise.

We initialize:

1. First column (dp[i][0]) is true → A sum of 0 is always possible using an empty subset.
2. First row (dp[0][j]) except dp[0][0] is false → With 0 elements, no sum > 0 is possible.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Understanding Inclusion and Exclusion

For each element arr[i] in our given set and each possible sum j, we have two choices:

1. Exclude the current element (arr[i])
   - This means we ignore arr[i] and check whether we could already achieve j using previous elements.
   - This is equivalent to dp[i-1][j] (i.e., check if we could form sum j without arr[i]).

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

2. Include the current element (arr[i])

- This means we use arr[i] to contribute to j.
- But to do this, we must check whether it was possible to form j - arr[i] before considering arr[i].
- This is equivalent to dp[i-1][j - arr[i]].
- If dp[i-1][j - arr[i]] == true, it means we can form j by adding arr[i].

The final result is determined by:

dp[i][j] = dp[i−1][j] (exclude) OR dp[i−1][j−arr[i]] (include)

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing the Element 3

We now process the first element 3, updating the table.

Logic for each j (target sum)

For each sum j, we decide:

- If we exclude 3 → Copy value from the row above.
- If we include 3 → Check if dp[i-1][j - 3] was true.

Since we only have one element (3), it can only contribute to sum 3.

Thus, dp[1][3] = true.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Sum ↓	{3}
0	✅
1	❌
2	❌
3	✅ (Using 3)
4	❌
5	❌
6	❌
7	❌
8	❌
9	❌

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing element 34 (No effect as it's greater than 9)

Sum ↓	{3, 34}
0	✅
1	❌
2	❌
3	✅ (From previous row)
4	❌
5	❌
6	❌
7	❌
8	❌
9	❌

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing the Element 4

We now incorporate 4 into our subset calculations.

Logic for each j (target sum)

For each sum j, we decide:

- If we exclude 4 → Copy the value from the row above.
- If we include 4 → Check if dp[i-1][j - 4] was true.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Updating the Table for 4

We now update the table by checking every sum from 1 to 9.

- j = 1:
  - Exclude 4 → ❌
  - Include 4? Not possible (since 4 > 1).
  - Final Result: dp[2][1] = false.
- j = 2:
  - Exclude 4 → ❌
  - Include 4? Not possible (since 4 > 2).
  - Final Result: dp[2][2] = false.
- j = 3:
  - Exclude 4 → ✅ (from 3).
  - Include 4? Not needed, as we already have 3.
  - Final Result: dp[2][3] = true ✅.
- j = 4:
  - Exclude 4 → ❌.
  - Include 4 → ✅ (since 4 itself forms the subset).
  - Final Result: dp[2][4] = true ✅.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

- j = 5:
  - Exclude 4 → ❌.
  - Include 4? Not possible (since dp[1][5 - 4] = dp[1][1] = ❌).
  - Final Result: dp[2][5] = false.
- j = 6:
  - Exclude 4 → ❌.
  - Include 4? Not possible (since dp[1][6 - 4] = dp[1][2] = ❌).
  - Final Result: dp[2][6] = false.
- j = 7:
  - Exclude 4 → ❌.
  - Include 4 → ✅ (dp[1][7 - 4] = dp[1][3] = ✅).
  - Final Result: dp[2][7] = true ✅.
- j = 8:
  - Exclude 4 → ❌.
  - Include 4? Not possible (since dp[1][8 - 4] = dp[1][4] = ❌).
  - Final Result: dp[2][8] = false.
- j = 9:
  - Exclude 4 → ❌.
  - Include 4? Not possible (since dp[1][9 - 4] = dp[1][5] = ❌).
  - Final Result: dp[2][9] = false.

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing element 4

Sum ↓	{3, 34, 4}
0	✅
1	❌
2	❌
3	✅
4	✅ (Using 4)
5	❌
6	❌
7	✅ (Using 3 & 4)
8	❌
9	❌

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing element 12 (No effect as it's greater than 9)

Sum ↓	{3, 34, 4, 12}
0	✅
1	❌
2	❌
3	✅
4	✅
5	❌
6	❌
7	✅
8	❌
9	❌

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Processing element 2

Sum ↓	{3, 34, 4, 12, 5, 2}
0	✅
1	❌
2	✅ (Using 2)
3	✅
4	✅
5	✅
6	✅ (Using 4 & 2)
7	✅
8	✅
9	✅ (Already True)

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Step 3: Extract the Answer

The final cell dp[6][9] is ✅ (True), meaning a subset exists that sums to 9.

Subset Example

One valid subset: {4, 5}

SUBSET SUM PROBLEM USING DYNAMIC PROGRAMMING

Elements → Target ↓	∅	3	34	4	12	5	2
Sum = 0	✅	✅	✅	✅	✅	✅	✅
Sum = 1	❌	❌	❌	❌	❌	❌	❌
Sum = 2	❌	❌	❌	❌	❌	✅	✅
Sum = 3	❌	✅	✅	✅	✅	✅	✅
Sum = 4	❌	❌	❌	✅	✅	✅	✅
Sum = 5	❌	❌	❌	❌	❌	✅	✅
Sum = 6	❌	❌	❌	✅	✅	✅	✅
Sum = 7	❌	❌	❌	❌	❌	✅	✅
Sum = 8	❌	❌	❌	✅	✅	✅	✅
Sum = 9	❌	❌	❌	✅	✅	✅	✅

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

The Longest Palindromic Subsequence (LPS) problem involves finding the longest subsequence in a given string that is a palindrome. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

For example:

- Input: "bbbab"
- Output: 4 (LPS is "bbbb")

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

Dynamic Programming Approach

We use Dynamic Programming (DP) to solve the problem by defining dp[i][j] as:

- dp[i][j] represents the length of the longest palindromic subsequence in the substring s[i:j].

Recurrence Relation:

1. Base Case:
    - A single character is always a palindrome → dp[i][i] = 1.
2. If the first and last characters match (s[i] == s[j]):
    - The LPS can extend beyond s[i+1:j-1].
    - Formula:

$$dp[i][j]=dp[i+1][j-1]+2$$

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

If the first and last characters do not match (s[i] != s[j]):

- The LPS is the maximum of either:
    - Excluding s[i] (dp[i+1][j])
    - Excluding s[j] (dp[i][j-1])
- Formula:

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

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

"BBBAB"

Step-by-Step Table Construction

Let's take an example:

Input: "bbbab"

String Length = n = 5

Step 1: Initialize dp Table

We initialize a 5x5 DP table where dp[i][i] = 1

(since a single character is always a palindrome).

i\j	0	1	2	3	4
0	1	0	0	0	0
1	-	1	0	0	0
2	-	-	1	0	0
3	-	-	-	1	0
4	-	-	-	-	1

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

Step 2: Fill Table for Length 2

- Compare adjacent characters.

Explanation:

- s[0] == s[1] (b == b) → dp[0][1] = 2
- s[1] == s[2] (b == b) → dp[1][2] = 2
- s[2] == s[3] (b == b) → dp[2][3] = 2
- s[3] != s[4] (b != a) → dp[3][4] =

  max(dp[4][4], dp[3][3]) = 1

"BBBAB"

i\j	0	1	2	3	4
0	1	2	0	0	0
1	-	1	2	0	0
2	-	-	1	2	0
3	-	-	-	1	1
4	-	-	-	-	1

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

Step 3: Fill Table for Length 3

Now, we consider substrings of length 3.

Explanation:

- s[0] == s[2] (b == b) → dp[0][2] = dp[1][1] + 2 = 3
- s[1] == s[3] (b == b) → dp[1][3] = dp[2][2] + 2 = 3
- s[2] != s[4] (b != a) → dp[2][4] = max(dp[3][4], dp[2][3]) = 2

"BBBAB"

i\j	0	1	2	3	4
0	1	2	3	0	0
1	-	1	2	3	0
2	-	-	1	2	2
3	-	-	-	1	1
4	-	-	-	-	1

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

Step 4: Fill Table for Length 4

- s[0] == s[3] (b == b) → dp[0][3] = dp[1][2] + 2 = 4
- s[1] != s[4] (b != a) → dp[1][4] = max(dp[2][4], dp[1][3]) = 3

"BBBAB"

i\j	0	1	2	3	4
0	1	2	3	4	0
1	-	1	2	3	3
2	-	-	1	2	2
3	-	-	-	1	1
4	-	-	-	-	1

LONGEST PALINDROMIC SUBSEQUENCE (LPS)

"BBBAB"

Step 5: Fill Table for Full String

- s[0] != s[4] (b != a) → dp[0][4] = max(dp[1][4], dp[0][3]) = 4

Final Answer

The length of the longest palindromic subsequence is dp[0][4] = 4.

i\j	0	1	2	3	4
0	1	2	3	4	4
1	-	1	2	3	3
2	-	-	1	2	2
3	-	-	-	1	1
4	-	-	-	-	1

MAXIMUM SIZE SQUARE SUBMATRIX WITH ALL 1S

Problem Statement

Given a binary matrix (containing only 0s and 1s), find the size of the largest square submatrix that contains only 1s.

MAXIMUM SIZE SQUARE SUBMATRIX WITH ALL 1S

Approach using Dynamic Programming

We use a DP table (dp[][]) where:

- dp[i][j] represents the size of the largest square whose bottom-right corner is at (i, j).
- Transition Formula:

$$dp[i][j]=\min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1$$

  - If matrix[i][j] = 1, the square can be extended.
  - If matrix[i][j] = 0, no square can end at (i, j), so dp[i][j] = 0.
- Base Case:
  - If i == 0 or j == 0, dp[i][j] = matrix[i][j] (copy first row and column as they are).
- Final Answer:
  - The maximum value in dp[][] gives the size of the largest square

MAXIMUM SIZE SQUARE SUBMATRIX WITH ALL 1S

Example Input

Matrix:

1	1	0	1
1	1	1	1
0	1	1	1
1	1	1	1

Step-by-Step DP Table Construction

Step 1: Initialize dp[][]

Copy the first row and column as they are:

1	1	0	1
1	?	?	?
0	?	?	?
1	?	?	?

MAXIMUM SIZE SQUARE SUBMATRIX WITH ALL 1S

Step 2: Fill dp[][] Using the Formula

dp[i][j]=min(dp[i−1][j],dp[i][j−1],dp[i−1][j−1])+1

Final dp[][] Table

1	1	0	1
1	1	1	1
0	1	2	2
1	1	2	3

i, j	Matrix[i][j]	Formula Applied	dp[i][j]
(1,1)	1	min(1,1,0) + 1	1
(1,2)	1	min(1,0,0) + 1	1
(1,3)	1	min(0,1,0) + 1	1
(2,1)	1	min(1,0,0) + 1	1
(2,2)	1	min(1,1,1) + 1	2
(2,3)	1	min(1,1,1) + 1	2
(3,1)	1	min(0,1,0) + 1	1
(3,2)	1	min(1,1,1) + 1	2
(3,3)	1	min(2,2,1) + 1	2

✅ Maximum value = 3, meaning the largest square is of size 3×3.

MINIMUM COST PATH USING DYNAMIC PROGRAMMING

Problem Statement

Given an n × m grid (matrix) where each cell contains a positive integer representing the cost to traverse that cell, find the minimum cost path from the top-left (0,0) to the bottom-right (n-1, m-1). You can only move right or down.

MINIMUM COST PATH USING DYNAMIC PROGRAMMING

We use a DP table dp[][] where:

- dp[i][j] represents the minimum cost to reach cell (i, j).
- The state transition formula:

$$dp[i][j]=cost[i][j]+min(dp[i-1][j],dp[i][j-1])$$

  - If moving down, cost comes from dp[i-1][j]
  - If moving right, cost comes from dp[i][j-1]
  - Take the minimum of both and add cost[i][j].
- Base Case:
- dp[0][0] = cost[0][0] (starting point)
- dp[i][0] = dp[i-1][0] + cost[i][0] (first column)
- dp[0][j] = dp[0][j-1] + cost[0][j] (first row)
- Final Answer:
- dp[n-1][m-1] gives the minimum cost to reach the bottom-right cell.

MINIMUM COST PATH USING DYNAMIC PROGRAMMING

Example Input

Matrix:

1	3	1
1	5	1
4	2	1

Step-by-Step DP Table Construction

Step 1: Initialize dp[][]

Copy the first row and column:

1	4	5
2	?	?
6	?	?

MINIMUM COST PATH USING DYNAMIC PROGRAMMING

Step 2: Fill dp[][] Using the Formula

dp[i][j]=cost[i][j]+min(dp[i−1][j],dp[i][j−1])

i, j	cost[i][j]	Formula Applied	dp[i][j]
(1,1)	5	5 + min(4,2)	7
(1,2)	1	1 + min(5,7)	6
(2,1)	2	2 + min(7,6)	6
(2,2)	1	1 + min(6,6)	7

Final dp[][] Table

1	4	5
2	7	6
6	6	7

✅ Minimum Cost Path = 7

EDIT DISTANCE USING DYNAMIC PROGRAMMING

Problem Statement

Given two strings word1 and word2, find the minimum number of operations required to convert word1 to word2. The allowed operations are:

1. Insert a character
2. Delete a character
3. Replace a character

EDIT DISTANCE USING DYNAMIC PROGRAMMING

We use a DP table dp[i][j] where:

- dp[i][j] represents the minimum number of operations needed to convert the first i characters of word1 to the first j characters of word2.
- State Transition Formula:
    - If characters match (word1[i-1] == word2[j-1]), no operation is needed:
        - dp[i][j]=dp[i−1][j−1]
    - If characters don’t match, take the minimum of three operations:
        - dp[i][j]=1+min(dp[i−1][j],dp[i][j−1],dp[i−1][j−1])
    - Delete (dp[i-1][j]) → Remove a character from word1
    - Insert (dp[i][j-1]) → Add a character to word1
    - Replace (dp[i-1][j-1]) → Change a character in word1

EDIT DISTANCE USING DYNAMIC PROGRAMMING

Example

Input:

word1 = "horse"

word2 = "ros"

EDIT DISTANCE USING DYNAMIC PROGRAMMING

Step-by-Step DP Table Construction

1. Initialize dp[][] Table
    - dp[i][0] = i → Deleting all characters in word1
    - dp[0][j] = j → Inserting all characters of word2

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

EDIT DISTANCE USING DYNAMIC PROGRAMMING

Fill dp[][] Using Formula

i, j	word1[i-1]	word2[j-1]	Operations	Formula Applied	dp[i][j]
(1,1)	h	r	Replace	1 + min(0,1,1)	1
(1,2)	h	o	Insert	1 + min(1,2,1)	2
(1,3)	h	s	Insert	1 + min(2,3,2)	3
(2,1)	o	r	Delete	1 + min(1,2,1)	2
(2,2)	o	o	No change	dp[1][1]	1
(2,3)	o	s	Insert	1 + min(1,3,2)	2
(3,1)	r	r	No change	dp[2][0]	2
(3,2)	r	o	Replace	1 + min(2,1,2)	2
(3,3)	r	s	Replace	1 + min(2,2,1)	2

EDIT DISTANCE USING DYNAMIC PROGRAMMING

Fill dp[][] Using Formula

i, j	word1[i-1]	word2[j-1]	Operations	Formula Applied	dp[i][j]
(4,1)	s	r	Delete	1 + min(2,3,2)	3
(4,2)	s	o	Delete	1 + min(3,2,2)	3
(4,3)	s	s	No change	dp[3][2]	2
(5,1)	e	r	Delete	1 + min(3,4,3)	4
(5,2)	e	o	Delete	1 + min(4,3,3)	4
(5,3)	e	s	Replace	1 + min(4,2,3)	3

EDIT DISTANCE USING DYNAMIC PROGRAMMING

		r	o	s
	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

✅ Minimum Edit Distance = dp[5][3] = 3

Operations:

1. Replace "h" → "r"
2. Delete "e"
3. Delete "o"