Dynamic Programming

Questions & Appreaches Taken from GeeksforGeeks

(Mostly focus on Top Down Approach - By creating 2D DP[][] array.)

1. Floyd Warshal Algorithm
2. Longest Common Subsequence &
3. DP - Different Question Types Notes
4. Longest common Subsequence (GeeksforGeeks)
5. Print LCS (GeeksforGeeks)
6. Edit Distance (Appln. of LCS) (GeeksforGeeks)
7. Length & Longest Increasing Subsequence (GFG)
8. Longest Common Substring (GFG)
9. Subset Sum (GeeksforGeeks)
10. Partition Problem (Appln of subset sum) (GFG)
11. 0/1 Knapsack (GFG)
12. Unbounded Knapsack (Appln of 0/1 Knapsack) (GFG)
13. Rod cutting Problem (Appln of 0/1 knapsack) (GFG)

Dynamic Programming

this tutorial, you will learn what dynamic programming is. Also, you will find the omparison between dynamic programming and greedy algorithms to solve roblems.

ynamic Programming is a technique in computer programming that helps to iciently solve a class of problems that have overlapping subproblems and ptimal substructure (https://en.wikipedia.org/wiki/Optimal_substructure) operty.

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

ich problems involve repeatedly calculating the value of the same subproblems to nd the optimum solution.

ynamic Programming Example

t's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence numbers in which each number is the sum of the two preceding ones. For ample, `0,1,1, 2, 3`. Here, each number is the sum of the two preceding mbers.

lgorithm

```
Let n be the number of terms.

1. If n <= 1, return 1.
2. Else, return the sum of two preceding numbers.
```

We are calculating the fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. `1 + 1 = 2`.
5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. `2 + 1 = 3`.

Hence, we have the sequence `0,1,1, 2, 3`. Here, we have used the results of the previous steps as shown below. This is called a **dynamic programming approach.**

```
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0)
F(3) = F(2) + F(1)
F(4) = F(3) + F(2)
```

How Dynamic Programming Works

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

```
var m = map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] = fib(n - 1) + fib(n - 2)
    return m[n]
```

Dynamic programming by memoization is a top-down approach to dynamic programming. By reversing the direction in which the algorithm works i.e. by starting from the base case and working towards the solution, we can also implement dynamic programming in a bottom-up manner.

```
function fib(n)
    if n = 0
        return 0
    else
        var prevFib = 0, currFib = 1
        repeat n - 1 times
            var newFib = prevFib + currFib
            prevFib = currFib
            currFib  = newFib
    return currentFib
```

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms (/dsa/greedy-algorithm) are similar to dynamic programming in the sense that they are both tools for optimization.

However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Different Types of Dynamic Programming Algorithms

1. Longest Common Subsequence (/dsa/longest-common-subsequence)
2. Floyd-Warshall Algorithm (/dsa/floyd-warshall-algorithm)

Floyd-Warshall Algorithm

In this tutorial, you will learn how floyd-warshall algorithm works. Also, you will find working examples of floyd-warshall algorithm in C, C++, Java and Python.

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

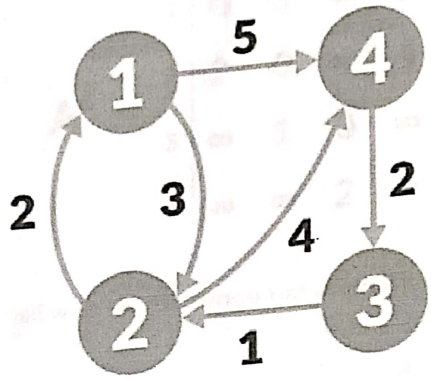
> A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming (/dsa/dynamic-programming) approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Initial graph

	1	2	3	4
1	0	3	7	5
2	2	0	6	4
3	3	1	0	5
4	5	3	2	0

← By Just directly seeing

Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix `A⁰` of dimension `n*n` where n is the number of vertices. The row and the column are indexed as `i` and `j` respectively. `i` and `j` are the vertices of the graph.

   Each cell `A[i][j]` is filled with the distance from the `iᵗʰ` vertex to the `jᵗʰ` vertex. If there is no path from `iᵗʰ` vertex to `jᵗʰ` vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Fill each cell with the distance between ith and jth vertex

2. Now, create a matrix `A¹` using matrix `A⁰`. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let `k` be the intermediate vertex in the shortest path from source to destination. In this step, `k` is the first vertex. `A[i][j]` is filled with `(A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j])`.

That is, if the direct distance from the source to the destination is greater than the path through the vertex `k`, then the cell is filled with `A[i][k] + A[k][j]`.

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Means vertex 1 as medium via.

Calculate the distance from the source vertex to destination vertex through this vertex k

For example: For `A¹[2, 4]`, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since `4 < 7`, `A⁰[2, 4]` is filled with 4.

3. Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2.**

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ \infty & & & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, A^3 and A^4 is also created.

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & \infty & \\ & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

Calculate the distance from the source vertex to destination vertex through this vertex 3

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \end{array}$$

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j])
return A
```

Python, Java and C/C++ Examples

Python | Java | C | C++

Floyd Warshall Algorithm Complexity

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

- To find the shortest path is a directed graph

File: /home/vicky/Desktop/FloydWarshall.java

```
package com.company;// Floyd Warshall Algorithm in Java

class FloydWarshall {
    final static int INF = 9999, nV = 4;

    // Implementing floyd warshall algorithm
    void floydWarshall(int graph[][]) {
        int matrix[][] = new int[nV][nV];
        int i, j, k;

        for (i = 0; i < nV; i++)
            for (j = 0; j < nV; j++)
                matrix[i][j] = graph[i][j];

        // Adding vertices individually
        for (k = 0; k < nV; k++) {
            for (i = 0; i < nV; i++) {
                for (j = 0; j < nV; j++) {
                    if (matrix[i][k] + matrix[k][j] < matrix[i][j])
                        matrix[i][j] = matrix[i][k] + matrix[k][j];
                }
            }
        }
        printMatrix(matrix);
    }

    void printMatrix(int matrix[][]) {
        for (int i = 0; i < nV; ++i) {
            for (int j = 0; j < nV; ++j) {
                if (matrix[i][j] == INF)
                    System.out.print("INF ");
                else
                    System.out.print(matrix[i][j] + "  ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int graph[][] = { { 0, 3, INF, 5 }, { 2, 0, INF, 4 }, { INF, 1, 0, INF },
{ INF, INF, 2, 0 } };
        FloydWarshall a = new FloydWarshall();
        a.floydWarshall(graph);
    }
}

/*** Output

0  3  7  5
2  0  6  4
3  1  0  5
5  3  2  0

*/
```

write this in LeetCode AIO & tear this note down not required much

taken as 'infinity' value

make matrix to do dp

this is what we are doing in Algo here

want to go from i vertex to j vertex via k vertex

understood

Just this is the main logic- not so tough to understand-

if i=j=2 then ans will be always '0' As same vertex -

Representation of Graph if 4 row (vertices) has what edges with other vertices (v=4)

DP on Graph

Question is: Given

Problem:- Given a graph with some vertices and directed weighted edges b/w them. Find the shortest path from one vertex to

Longest Common Subsequence

In this tutorial, you will learn how the longest common subsequence is found. Also, you will find working examples of the longest common subsequence in C, C++, Java and Python.

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.

If `S1` and `S2` are the two given sequences then, `Z` is the common subsequence of `S1` and `S2` if `Z` is a subsequence of both `S1` and `S2`. 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`.

If

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

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

Let us understand LCS with an example.

If

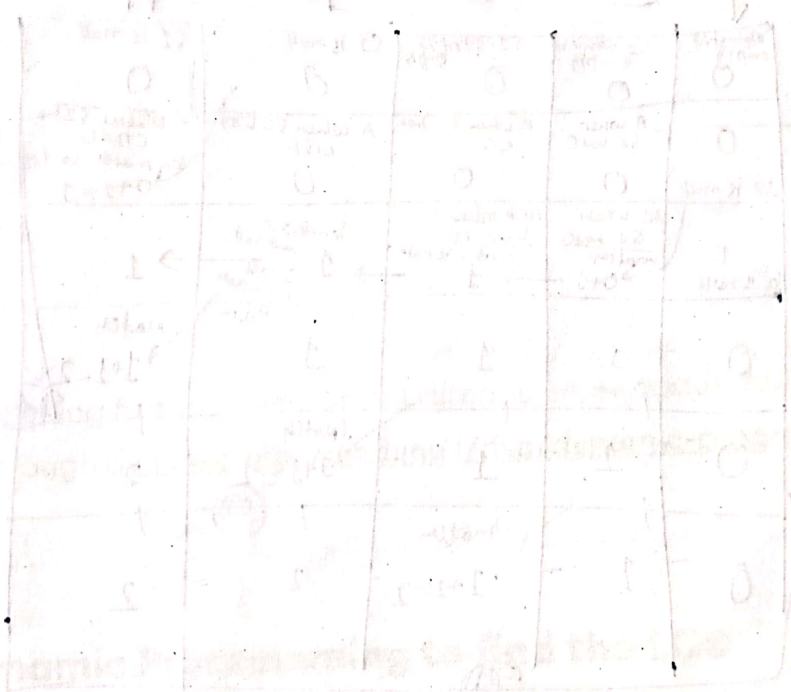
```
S1 = {B, C, D, A, A, C, D}
S2 = {A, C, D, B, A, C}
```

Then, common subsequences are

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

Make common.

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



We are going to find this longest common subsequence using dynamic programming.

Before proceeding further, if you do not already know about dynamic programming, please go through dynamic programming (/dsa/dynamic-programming).

Using Dynamic Programming to find the LCS

Let us take two sequences:

X

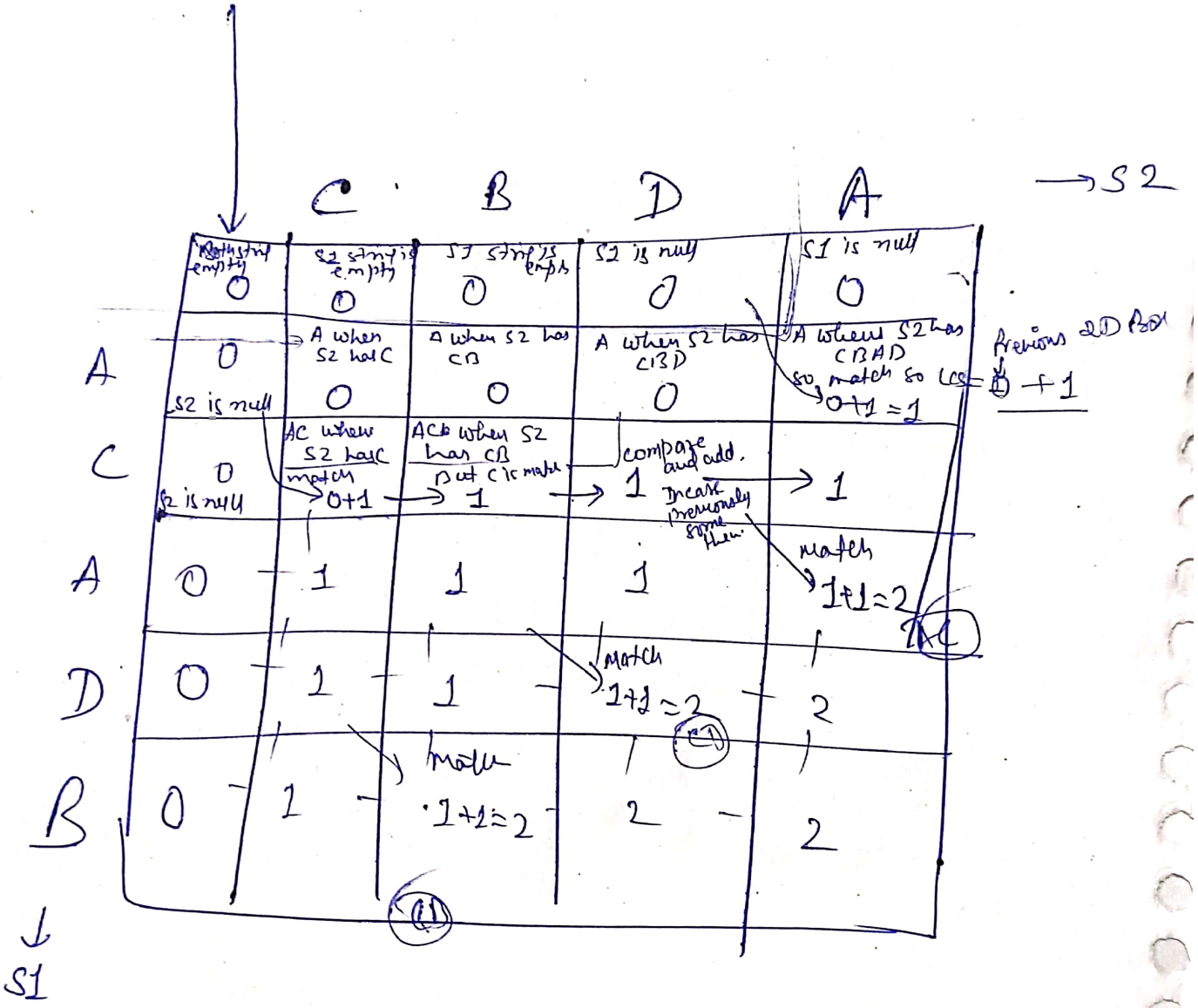
A	C	A	D	B

The first sequence

Y

C	B	D	A

Second Sequence


C
B
D
A
→ S2
S1 is null
A when S2 has CBAD so match so 0+1=1
Previous DD Box + 1
A when S2 has CB
A when S2 has CBD
AC when S2 has C match 0+1
ACB when S2 has CB but C is match
compare and add. Increase previously stored value
match 1+1=2
match 1+1=2
match 1+1=2
S2 is null
S1 is null
A
C
A
D
B
↓ S1

1. Create a table of dimension `n+1*m+1` where n and m are the lengths of `X` and `Y` respectively. The first row and the first column are filled with zeros.

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

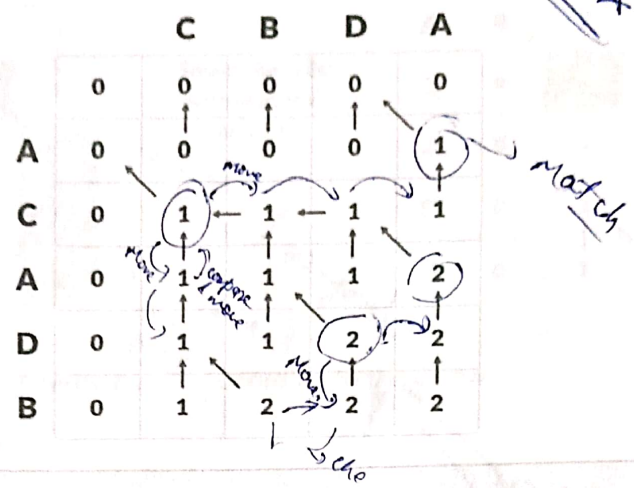
Initialise a table

2. Fill each cell of the table using the following logic.
3. 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.
4. 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.

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

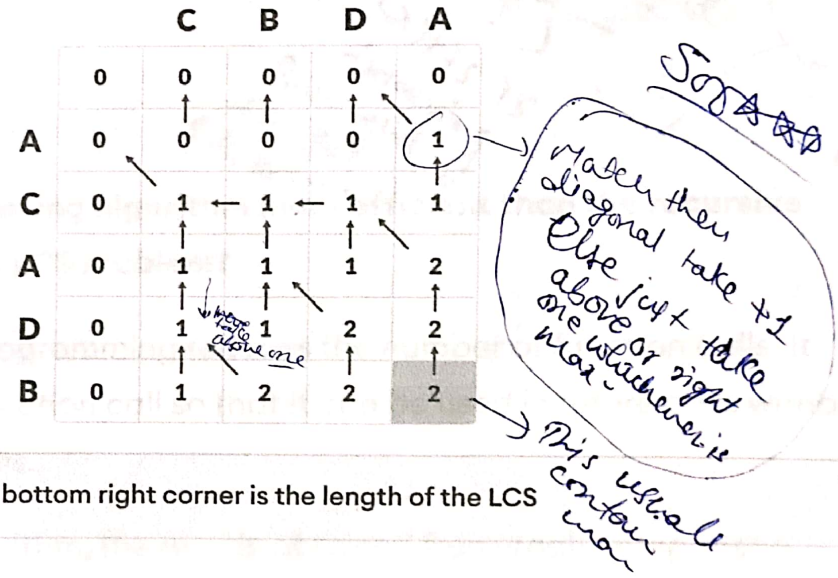
Fill the values

5. **Step 2** is repeated until the table is filled.



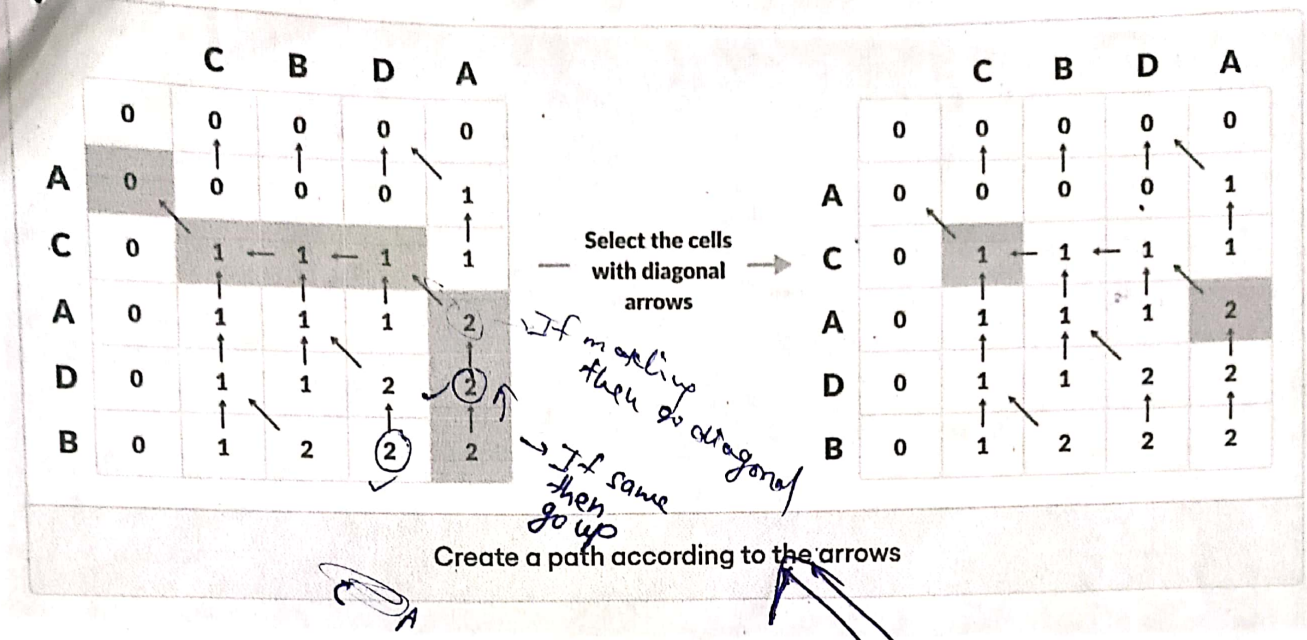
Fill all the values

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



The bottom right corner is the length of the LCS

7. 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.



Thus, the longest common subsequence is `CA`.

C	A

LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of `X` and the elements of `Y` are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of $2^{max(m, n)}$.

Longest Common Subsequence Algorithm

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Python, Java and C/C++ Examples

Python | Java | C | C++

Longest Common Subsequence Applications

1. In compressing genome resequencing data
2. to authenticate users within their mobile phone through in-air signatures

Next Tutorial:
Backtracking Algorithm →(/dsa/backtracking-algorithm)

Previous Tutorial:
Floyd-Warshall Algorithm (/dsa/floyd-warshall-algorithm)

Make crisp code notes & approach for this to understand

File: /home/vicky/Desktop/LCS_ALGO.java

```
package com.company;// The longest common subsequence in Java

class LCS_ALGO {
    static void lcs(String S1, String S2, int m, int n) {
        int[][] LCS_table = new int[m + 1][n + 1];

        // Building the mtrix in bottom-up way
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0)
                    LCS_table[i][j] = 0;
                else if (S1.charAt(i - 1) == S2.charAt(j - 1))
                    LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
                else
                    LCS_table[i][j] = Math.max(LCS_table[i - 1][j], LCS_table[i][j
 - 1]);
            }
        }
        //Here we are getting that subsequence.
        int index = LCS_table[m][n];
        int temp = index;   // 2 since max values

        char[] lcs = new char[index + 1];
        lcs[index] = '\0';

        int i = m, j = n;   // start from end of matrix
        while (i > 0 && j > 0) {
            if (S1.charAt(i - 1) == S2.charAt(j - 1)) {
                lcs[index - 1] = S1.charAt(i - 1);

                i--;
                j--;
                index--;
            }

            else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
                i--;
            else
                j--;
        }

        // Printing the sub sequences
        System.out.print("S1 : " + S1 + "\nS2 : " + S2 + "\nLCS: ");
        for (int k = 0; k <= temp; k++)
            System.out.print(lcs[k]);
        System.out.println("");
    }

    public static void main(String[] args) {
        String S1 = "ACADB";
        String S2 = "CBDA";
        int m = S1.length();
        int n = S2.length();
        lcs(S1, S2, m, n);
    }
}

/*** Output


 S1 : ACADB
 S2 : CBDA
 LCS: CB

 */
```

- Ok understood approach & code
- when char match then go back diagonal
- So it will only store two char even we can come across more
- Compare up hand side & key and whichever is max go to that either back or leftside up
- Left same in left & up then go up case is not mentioned here.
- It's wrong but concept is left to down.
- This is reverse traversing of our matrix to get subsequence string.
- This will go back diagonally first values
- Start from end of matrix

Himanshu Singour • 2nd
Specialist Programmer SWE At @Infosys CP - 4★Leetcode | 5...
2w • Edited •

+ Follow •••

Dynamic Programming All Patterns Preparation Bst

Pattern ->1D Dynamic Programming

1) Climbing Stairs
2) Frog Jump
3) Frog Jump with K
4) Maximum sum of Non-adjacent elements
5) House robber 2
6) Ninja's Training

Pattern -> Dynamic Programming On Grids / 2D

1) Grid Unique Paths
2) Unique Paths
3) Minimum path sum in Grid
4) Tringle (Fixed Starting Point and Variable Ending Point)
5) Minimum/Maximum Falling Path Sum
6) Cherry Pickup 2

Pattern -> Dynamic Programming On Subsets / Subsequence

1) Subset Sum Equals to Target
2) Partition Equals Subset Sum
3) Partition A subset into 2 subset with minimum absolute sum diff.
4) Count Subsets With Sum K
5) Count Partitions with given difference
6) 0/1 Knapsack
7) Minimum Coins
8) Target Sum
9) Coin Change 2
10) Unbounded Knapsack 1 D array
11) Rod Cutting 1 D array

Pattern -> Dynamic Programming On Strings

1) Print Length Of Longest Common Subsequence (LCS) → This is one which is partially used in every below mentioned questions)
2) Print Longest Common Subsequence (Use LCS)
3) Longest Palindromic Subsequence
4) Minimum Insertion to Make String Palindrome
5) Minimum Insertions/Deletions to Convert String A -> B
6) Shortest Common SuperSequence
7) Distinct Subsequence
8) Edit Distance (Uses LCS only)
9) Wildcard Matching

Pattern -> Dynamic Programming On Stocks

1) Best Time To Buy and Sell Stock (Buy Ones & Sell Ones)

SubArray {1, 2, 3, 4, 5}

↓

{2,3}, {2,3,4,5} {4,5} etc

substring S = ABCDEF

S1 = ABC, S2 = CDE, S3 = ABCDE etc

---

subsequence {1, 2, 3, 4, 5}

{1, 4, 5}, {2, 3, 5}, {2, 3, 4}, {1, 5} etc

{2,3,4} → subarray is a type of subsequence only.

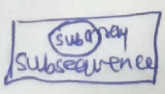
Subsequence S = ABCDEF

S1 = ACF, S2 = Af, S3 = BEF,

S4 = ABF, S5 = DEF etc.

DEF → subarray is also can be subsequence subarray is

Understand by Set Diagram



1) Best Time To Buy and Sell Stock 2 (Unlimited Time Buy & Sell)
1) Best Time To Buy and Sell Stock 3 (At Max 2 Times Buy & Sell)
1) Best Time To Buy and Sell Stock 4 (K times Buy & Sell)
1) Best Time To Buy and Sell Stock 5 (Buy & Sell With Cooldown)
1) Best Time To Buy and Sell Stock 6 (Buy & Sell With Extra Fee)

Pattern -> **Dynamic Programming On Longest Increasing Subsequence** (LIS)
1) Print Length Of Longest Increasing Subsequence
2) Print Longest Increasing Subsequence
3) Largest Divisible Subset
4) Longest String chain
5) Longest Bitonic Subsequence
6) Number Of Longest Increasing Subsequence

Pattern -> **Hardest Dynamic Programming On partition**
1) Matrix Chain Multiplication
2) Minimum Cost To cut The Stick
3) Burst Balloons (Asked in samsung, I remember)
4) Evaluate Boolean
5) Palindrome Partitioning 2
6) Partition Array For Maximum Sum
7) Maximum Rectangle Area with all 1's (Dp on Rectangle)
8) Count Square Submatrices with all ones (Dp on Rectangle)

"I Think These All Questions Are Enough To Mastering Dynamic programming"

You will find all the solutions and explanations on this channel **takeUforward by Raj Vikramaditya** bhaiya

#dsa #interviewprepration #dp #dynamicprogramming #algorithum

Pushkar Raja and 1,615 others 26 comments • 32 shares

Reactions

Love this... I wonder... This will help me... Thank you for...

Like Comment Share Send

Add a comment...

Most relevant

Raj Vikramaditya • 2nd 2w (edited) •••
Software Engineer @ Google | YouTuber(200k+) | Ex-Media.net, Amazon | JGEC

Top 20 Dynamic Programming Interview Questions

Difficulty Level : Hard • Last Updated : 22 Jun, 2022

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

If Any of below question you find it difficult to understand the code can always first check the Youtube videos.

Following are the most important Dynamic Programming problems asked in various Technical Interviews.

1. Longest Common Subsequence
2. Longest Increasing Subsequence
3. Edit Distance
4. Minimum Partition (Before this do Partition Problem (DP-18) (Added here also in notes) Also Do subset sum DP Problem)
5. Ways to Cover a Distance
6. Longest Path In Matrix
7. Subset Sum Problem (Print out your code written)
8. Optimal Strategy for a Game
9. 0-1 Knapsack Problem
10. Boolean Parenthesization Problem
11. Shortest Common Supersequence
12. Matrix Chain Multiplication
13. Partition problem
14. Rod Cutting
15. Coin change problem
16. Word Break Problem
17. Maximal Product when Cutting Rope
18. Dice Throw Problem
19. Box Stacking
20. Egg Dropping Puzzle

2nd Same approach.

Please check your internet and try again.

Post Title

Longest Common Subsequence | DP-4

Paragraph

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively. We also discussed one example problem in Set 3. Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bdf", "aeg", '"acefg", .. etc are subsequences of "abcdefg".

In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall from theory of permutation and combination that number of combinations with 1 element are ${}^nC_1$. Number of combinations with 2 elements are ${}^nC_2$ and so forth and so on. We know that ${}^nC_0 + {}^nC_1 + {}^nC_2 + \ldots {}^nC_n = 2^n$. So a string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be $O(n * 2^n)$. Note that it takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:
LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.
LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

Recommended Practice
Longest Common Subsequence
Try It!
Div: practiceLinkDiv

Div: extraAd
The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:
Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = MAX (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))

Examples:
1) Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as:
L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

2) Consider the input strings "ABCDGH" and "AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:
L("ABCDGH", "AEDFHR") = MAX (L("ABCDG", "AEDFHR"), L("ABCDGH", "AEDFH"))
So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

2) Overlapping Subproblems: (To understand this is recursive in nature)
Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

C++ C Java Python3 C# PHP Javascript

```
/* A Naive recursive implementation of LCS problem in java*/
public class LongestCommonSubsequence
{

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char[] X, char[] Y, int m, int n )
{
    if (m == 0 || n == 0)
    return 0;
    if (X[m-1] == Y[n-1])
    return 1 + lcs(X, Y, m-1, n-1);
    else
    return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
```

Recursive Approach

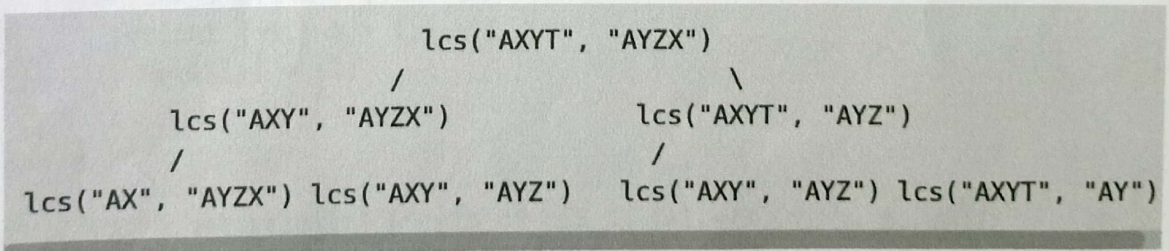
```
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

public static void main(String[] args)
{
    LongestCommonSubsequence lcs = new LongestCommonSubsequence();
    String s1 = "AGGTAB";
    String s2 = "GXTXAYB";

    char[] X=s1.toCharArray();
    char[] Y=s2.toCharArray();
    int m = X.length;
    int n = Y.length;

    System.out.println("Length of LCS is" + " " +
                                lcs.lcs( X, Y, m, n ) );
}

}

// This Code is Contributed by Saket Kumar
```

Output

```
Length of LCS is 4
```

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.

Considering the above implementation, following is a partial recursion tree for input strings "AXYT" and "AYZX"



In the above partial recursion tree, lcs("AXY", "AYZ") is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

Following is a Memoization implementation for the LCS problem.

C++ Java Python3 C# Javascript

```java
/*package whatever //do not write package name here */
import java.io.*;

class GFG
{
  // A Top-Down DP implementation of LCS problem

  // Returns length of LCS for X[0..m-1], Y[0..n-1]
  static int lcs(String X,String Y,int m,int n,int[][] dp){

    if (m == 0 || n == 0)
      return 0;

    if (dp[m][n] != -1)
      return dp[m][n];

    if(X.charAt(m - 1) == Y.charAt(n - 1)){
      dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);
      return dp[m][n];
    }

    dp[m][n] = Math.max(lcs(X, Y, m, n - 1, dp),lcs(X, Y, m - 1, n, dp));
    return dp[m][n];
  }

  // Drivers code
  public static void main(String args[]){

    String X = "AGGTAB";
    String Y = "GXTXAYB";

    int m = X.length();
    int n = Y.length();
    int[][] dp = new int[m + 1][n + 1];
    for(int i=0;i<m + 1;i++){
      for(int j=0;j<n + 1;j++){
        dp[i][j] = -1;
      }
    }

    System.out.println("Length of LCS is "+lcs(X, Y, m, n, dp));

  }
}

// This code is contributed by shinjanpatra
```

Output

```
Length of LCS is 4
```

Time Complexity : O(mn) ignoring recursion stack space

Following is a tabulated implementation for the LCS problem.

Python3 | C++ | Java | Python3 | C# | PHP | Javascript

```
/* Dynamic Programming Java implementation of LCS problem */
public class LongestCommonSubsequence
{

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char[] X, char[] Y, int m, int n )
{
    int L[][] = new int[m+1][n+1];

    /* Following steps build L[m+1][n+1] in bottom up fashion. Note
        that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for (int i=0; i<=m; i++)
    {
    for (int j=0; j<=n; j++)
    {
    if (i == 0 || j == 0)
            L[i][j] = 0;
    else if (X[i-1] == Y[j-1])
            L[i][j] = L[i-1][j-1] + 1;
     else
            L[i][j] = max(L[i-1][j], L[i][j-1]);
    }
    }
return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

public static void main(String[] args)
{
    LongestCommonSubsequence lcs = new LongestCommonSubsequence();
    String s1 = "AGGTAB";
    String s2 = "GXTXAYB";

    char[] X=s1.toCharArray();
    char[] Y=s2.toCharArray();
```

```
        int m = X.length;
        int n = Y.length;

        System.out.println("Length of LCS is" + " " +
                                    lcs.lcs( X, Y, m, n ) );
    }

}

// This Code is Contributed by Saket Kumar
```

Output

```
Length of LCS is  4
```

Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation.
The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.
Printing Longest Common Subsequence
You can also check the space optimized version of LCS at
Space Optimized Solution of LCS

Words: 754
Characters: 4814

S1 = "BCDAACD"

S2 = "ACDBAC"

→ Taken this much string in that DP[j]

Taken this much String at that DP[i] ↓

	Empty String	A	C	D	B	A	C
Empty String	0	0	0	0	0	0	0
B	0	0	0	0	1	1	1
C	0	0	1	1	1	1	2
D	0	0	1	2	2	2	2
A	0	1	1	2	2	3	3
A	0	1	1	2	2	3	3
C	0	1	2	2	2	3	4
D	0	1	2	3	3	3	4

match

Here why we not using previous row or previous column and add 1 to it bcoz we might end up using duplicate as you can see here

① If match, then get diagonal previous element and add 1 to it and put in DP array.

② If not match get max(dp[i-1][j], dp[i][j-1])

Post Title

Printing Longest Common Subsequence

Given two sequences, print the longest subsequence present in both of them.

Examples:

- LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.
- LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

We have discussed Longest Common Subsequence (LCS) problem in a previous post. The function discussed there was mainly to find the length of LCS. To find length of LCS, a 2D table L[][] was constructed. In this post, the function to construct and print LCS is discussed.

Following is detailed algorithm to print the LCS. It uses the same 2D table L[][].

1. Construct L[m+1][n+1] using the steps discussed in previous post.
2. The value L[m][n] contains length of LCS. Create a character array lcs[] of length equal to the length of lcs plus 1 (one extra to store \0).
3. Traverse the 2D array starting from L[m][n]. Do following for every cell L[i][j]
   - → If characters (in X and Y) corresponding to L[i][j] are same (Or X[i-1] == Y[j-1]), then include this character as part of LCS.
   - → Else compare values of L[i-1][j] and L[i][j-1] and go in direction of greater value.

The following table (taken from Wiki) shows steps (highlighted) followed by the above algorithm.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

If matching then in LCS char and go diago

Go in direction of greater value if not matching

start from here

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.Div: practiceLinkDiv

Following is the implementation of above approach.

C++14 | Java | Python3 | C# | PHP | Javascript

```
// Dynamic Programming implementation of LCS problem in Java
```

```
import java.io.*;

class LongestCommonSubsequence {
    // Returns length of LCS for X[0..m-1], Y[0..n-1]
    static void lcs(String X, String Y, int m, int n)
    {
        int[][] L = new int[m + 1][n + 1];

        // Following steps build L[m+1][n+1] in bottom up
        // fashion. Note that L[i][j] contains length of LCS
        // of X[0..i-1] and Y[0..j-1]
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0)
                    L[i][j] = 0;
                else if (X.charAt(i - 1) == Y.charAt(j - 1))
                    L[i][j] = L[i - 1][j - 1] + 1;

                else
                    L[i][j] = Math.max(L[i - 1][j],
                                       L[i][j - 1]);

            }
        }

        // Following code is used to print LCS
        int index = L[m][n];
        int temp = index;

        // Create a character array to store the lcs string
        char[] lcs = new char[index + 1];
        lcs[index]
            = '\u0000'; // Set the terminating character (These things are in C. Never seen in Java Language)

        // Start from the right-most-bottom-most corner and
        // one by one store characters in lcs[]
        int i = m;
        int j = n;
        while (i > 0 && j > 0) {
            // If current character in X[] and Y are same,
            // then current character is part of LCS
            if (X.charAt(i - 1) == Y.charAt(j - 1)) {
                // Put current character in result
                lcs[index - 1] = X.charAt(i - 1);

                // reduce values of i, j and index
                i--;
                j--;
                index--;
            }

            // If not same, then find the larger of two and
            // go in the direction of larger value
            else if (L[i - 1][j] > L[i][j - 1])
```

In Java strings don't terminate with '\0' char.

This is just same when ~~by~~ left & top are same as mentioned in Approach in back page.

Write

```
            i--;
        else
            j--;
    }

    // Print the lcs
    System.out.print("LCS of " + X + " and " + Y
                     + " is ");
    for (int k = 0; k <= temp; k++)
        System.out.print(lcs[k]);
}

// driver program
public static void main(String[] args)
{
    String X = "AGGTAB";
    String Y = "GXTXAYB";
    int m = X.length();
    int n = Y.length();
    lcs(X, Y, m, n);
}
}

// Contributed by Pramod Kumar
```

ok understood fully

Output

```
LCS of AGGTAB and GXTXAYB is GTAB
```

Time Complexity: O(m*n)
Auxiliary Space: O(m*n)

Words: 335
Characters: 1489

Post Title

Edit Distance | DP-5

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

1. Insert
2. Remove
3. Replace

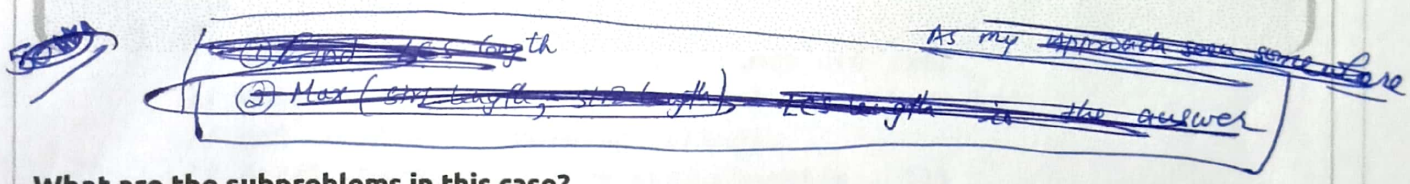
All of the above operations are of equal cost.

Examples:

Input: *str1 = "geek", str2 = "gesek"*
Output: *1*
Explanation: *We can convert str1 into str2 by inserting a 's'.*

Input: *str1 = "cat", str2 = "cut"*
Output: *1*
Explanation: *We can convert str1 into str2 by replacing 'a' with 'u'.*

Input: *str1 = "sunday", str2 = "saturday"*
Output: *3*
Explanation: *Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations. Replace 'n' with 'r', insert t, insert a*



What are the subproblems in this case?
The idea is to process all characters one by one starting from either from left or right sides of both strings.
Let us traverse from right corner, there are two possibilities for every pair of character being traversed.

```
m: Length of str1 (first string)
n: Length of str2 (second string)
```

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
    1. Insert: Recur for m and n-1
    2. Remove: Recur for m-1 and n
    3. Replace: Recur for m-1 and n-1

Below is implementation of above Naive recursive solution.

C++ Java Python3 C# PHP Javascript

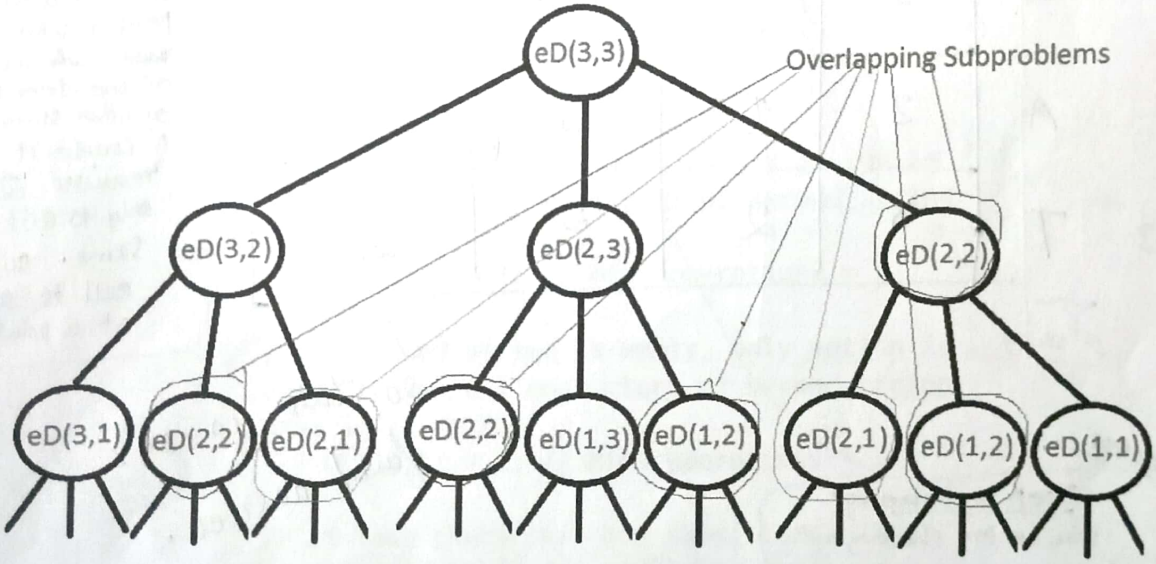
```
// A Naive recursive Java program to find minimum number
// operations to convert str1 to str2
class EDIST {
    static int min(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return x;
        if (y <= x && y <= z)
            return y;
        else
            return z;
    }

    static int editDist(String str1, String str2, int m,
                        int n)
    {
        // If first string is empty, the only option is to
        // insert all characters of second string into first
        if (m == 0)
            return n;

        // If second string is empty, the only option is to
        // remove all characters of first string
        if (n == 0)
            return m;

        // If last characters of two strings are same,
        // nothing much to do. Ignore last characters and
        // get count for remaining strings.
        if (str1.charAt(m - 1) == str2.charAt(n - 1))
            return editDist(str1, str2, m - 1, n - 1);

        // If last characters are not same, consider all
        // three operations on last character of first
        // string, recursively compute minimum cost for all
        // three operations and take minimum of three
        // values.
        return 1
            + min(editDist(str1, str2, m, n - 1), // Insert
                  editDist(str1, str2, m - 1, n), // Remove
                  editDist(str1, str2, m - 1,
                           n - 1) // Replace
              );
    }

    // Driver Code
    public static void main(String args[])
    {
        String str1 = "sunday";
```

```
        String str2 = "saturday";

        System.out.println(editDist(
            str1, str2, str1.length(), str2.length()));
    }
}
/*This code is contributed by Rajat Mishra*/
```

Output

```
3
```

The **time complexity** of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.
Auxiliary Space: O(1), because no extra space is utilized.



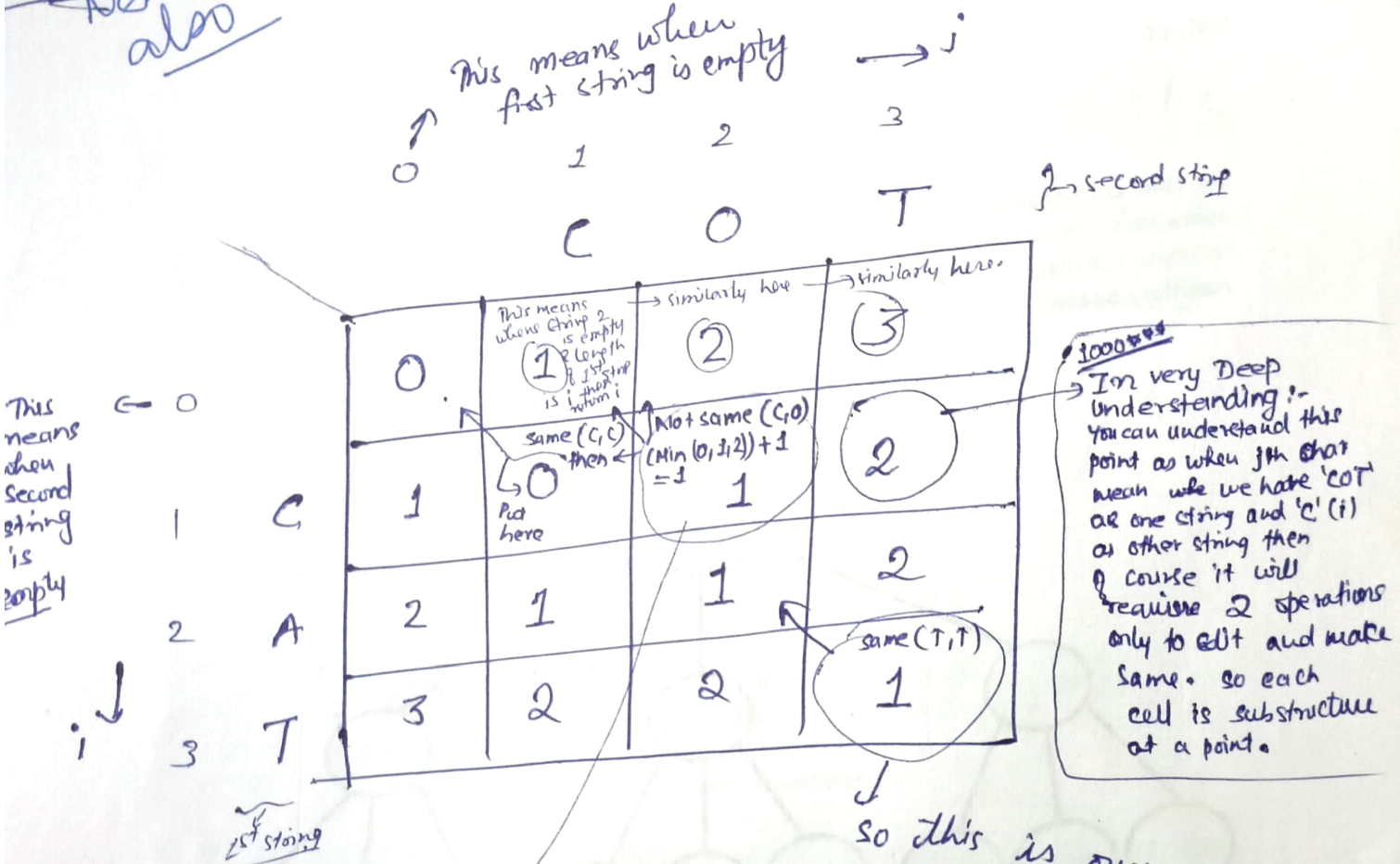
Worst case recursion tree when m = 3, n = 3.
Worst case example str1="abc" str2="xyz"

We can see that many subproblems are solved, again and again, for example, eD(2, 2) is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

100 %%%

Approach Explained for Edit

See next page also



1000%%%

Deep Understanding :-
Why we are choosing min^m, because we can come to this cell in 3 ways
→ ↘ ↓
So we ~~can~~ should be there using min^m operations only so

S1 = COT →
S2 = CAT →

1 operation → Replace O to A

Max(S1,S2) − LCS
5 − 3

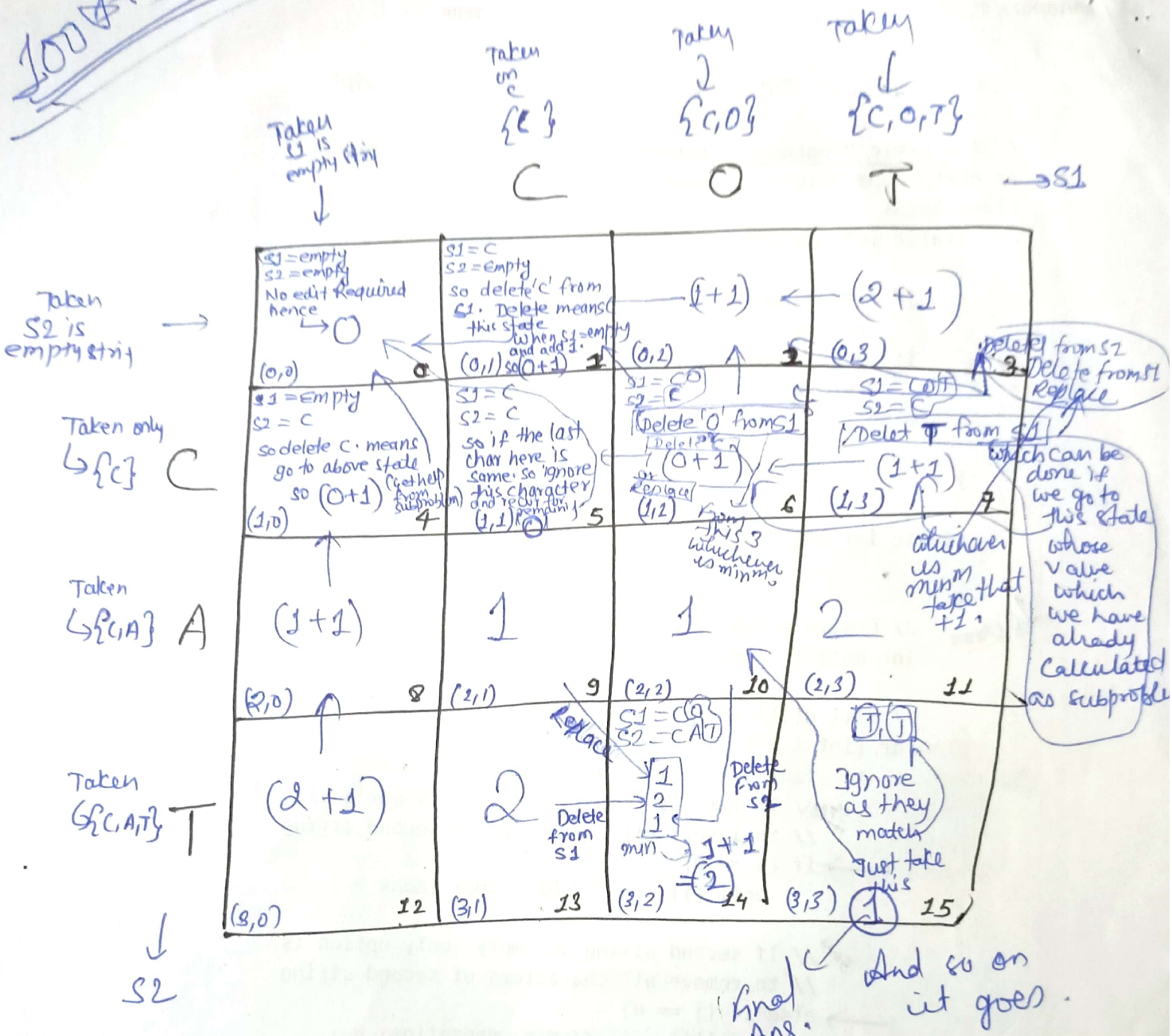
(Usually I've seen this way answer is generally coming.

CRATE
CAT
→ LCS ⇒ CAT {R, E} Remove (2 operations min^m)

CRAET → LCS → (CAT) {R, E} Remove (2 operations min^m)

C++ Java Python3 C# PHP Javascript

```
// A Dynamic Programming based Java program to find minimum
// number operations to convert str1 to str2
class EDIST {
    static int min(int x, int y, int z)
    {
        if (x <= y && x <= z)
            return x;
        if (y <= x && y <= z)
            return y;
        else
            return z;
    }

    static int editDistDP(String str1, String str2, int m,
                          int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m + 1][n + 1];

        // Fill d[][] in bottom up manner
        for (int i = 0; i <= m; i++) {
            for (int j = 0; j <= n; j++) {
                // If first string is empty, only option is
                // to insert all characters of second string
                if (i == 0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is
                // to remove all characters of second string
                else if (j == 0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last
                // char and recur for remaining string
                else if (str1.charAt(i - 1)
                         == str2.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];

                // If the last character is different,
                // consider all possibilities and find the
                // minimum
                else
                    dp[i][j] = 1
                               + min(dp[i][j - 1], // Insert
                                     dp[i - 1][j], // Remove
                                     dp[i - 1]
                                       [j - 1]); // Replace
            }
        }

        return dp[m][n];
    }
}
```



(1) Each cell Represents a Meaning

→ cell 0 Represents (0,0) → Taken both string S1 and S2 empty.

→ At any point we can either insert, Remove or Replace

→ cell 5 Represent (1,1) → Taken {C} from string 1 & {C} from string 2

Cell 11 Represent {2,3} → Taken {C,O,T} from string 1 & {CA} from string 2

```
        return dp[m][n];
    }

    // Driver Code
    public static void main(String args[])
    {
        String str1 = "sunday";
        String str2 = "saturday";
        System.out.println(editDistDP(
            str1, str2, str1.length(), str2.length()));
    }
} /*This code is contributed by Rajat Mishra*/
```

Output

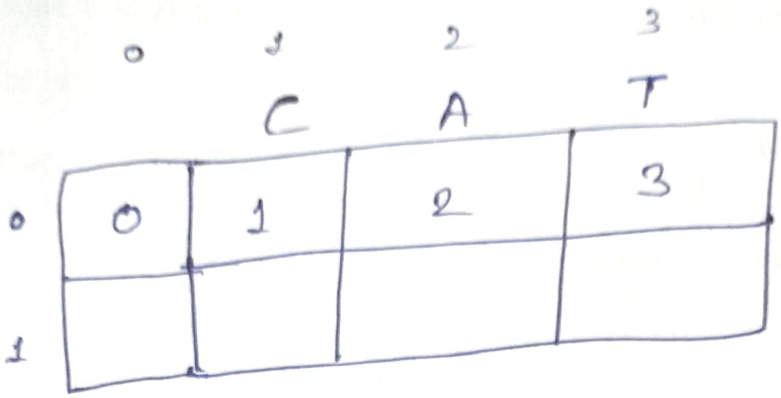
```
3
```

Time Complexity: O(m x n)
Auxiliary Space: O(m x n)

Space Complex Solution: In the above-given method we require O(m x n) space. This will not be suitable if the length of strings is greater than 2000 as it can only create 2D array of 2000 x 2000. To fill a row in DP array we require only one row the upper row. For example, if we are filling the i = 10 rows in DP array we require only values of 9th row. So we simply create a DP array of 2 x str1 length. This approach reduces the space complexity. Here is the C++ implementation of the above-mentioned problem

C++ | Java | Python3 | C# | Javascript

```
// A Space efficient Dynamic Programming
// based Java program to find minimum
// number operations to convert str1 to str2
import java.util.*;
class GFG
{

static void EditDistDP(String str1, String str2)
{
    int len1 = str1.length();
    int len2 = str2.length();

    // Create a DP array to memoize result
    // of previous computations
    int [][]DP = new int[2][len1 + 1];


    // Base condition when second String
```



⇒ when i=0

separately done ⇒

str1 = CAT

str2 = COT

---

for: i=1 to len1+1

i< len2
j< len1

Rest operations condition Remains same as in mxn array like only.

only to flip arr1 & arr2 we use concept of dp[i%2]

arr1 → At a time, it is being used to update arr2 and other time vice versa using concept of dp[i%2][j]

```
        // is empty then we remove all characters
        for (int i = 0; i <= len1; i++)

            DP[0][i] = i;

        // Start filling the DP
        // This loop run for every
        // character in second String
        for (int i = 1; i <= len2; i++)
        {

            // This loop compares the char from
            // second String with first String
            // characters
            for (int j = 0; j <= len1; j++)
            {

                // if first String is empty then
                // we have to perform add character
                // operation to get second String
                if (j == 0)
                    DP[i % 2][j] = i;

                // if character from both String
                // is same then we do not perform any
                // operation . here i % 2 is for bound
                // the row number.
                else if (str1.charAt(j - 1) == str2.charAt(i - 1)) {
                    DP[i % 2][j] = DP[(i - 1) % 2][j - 1];
                }

                // if character from both String is
                // not same then we take the minimum
                // from three specified operation
                else {
                    DP[i % 2][j] = 1 + Math.min(DP[(i - 1) % 2][j],
                                                Math.min(DP[i % 2][j - 1],
                                                         DP[(i - 1) % 2][j - 1]));

                }
            }
        }

        // after complete fill the DP array
        // if the len2 is even then we end
        // up in the 0th row else we end up
        // in the 1th row so we take len2 % 2
        // to get row
        System.out.print(DP[len2 % 2][len1] +"\n");
    }

    // Driver program
    public static void main(String[] args)
    {
```

Separately do for i=0

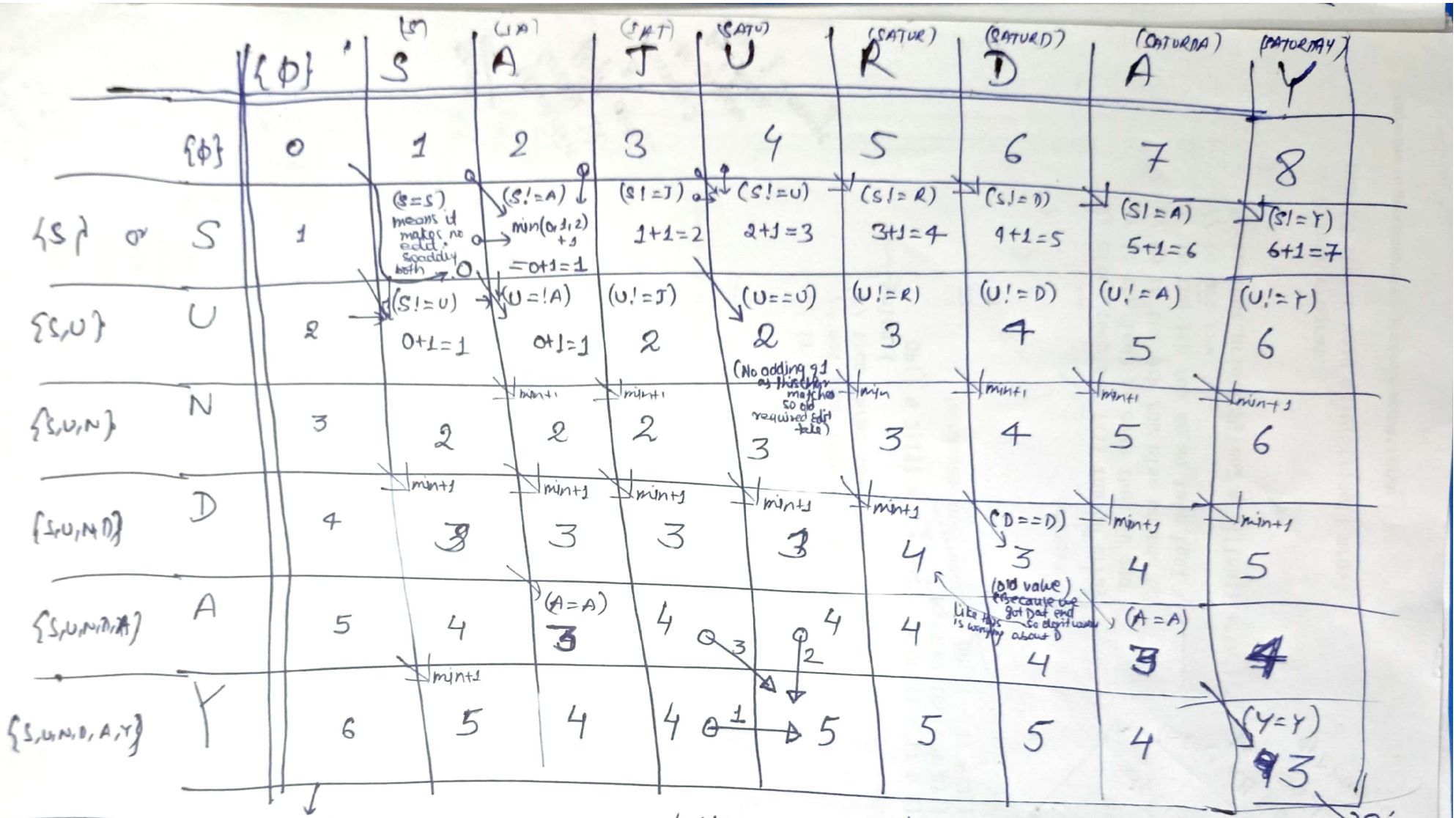
Only this extra condition & flipping array is done.

So ppt

Using this concept it flips array 1 & array 2 to compare and make use of 2 arrays

This conditions & updation are same.

Overall here just change dp[i][ ] → to → dp[i%2][ ] Rest all same as previous bottom up approach only



Understanding Replacing

↳ Replace is like removing this both. Then what ~~not~~ was it's answer, like asking what was it's answer when this both mismatch char are not taken then so ask to that sub-problem → so replace is exactly like that

Understanding the arrows mentioned above

Arrow 1 → (j-1) → It is saying in our string we want to Add "U"

Arrow 2 → (i-1) → It is saying in our string we want to Remove "Y"

Arrow 3 → (i-1)(j-1) → It is saying like it Replace U to Y or Y to U (similarly kind) It also looks like adding with or Replace deleting both or [illegible]

```
    String str1 = "food";
    String str2 = "money";

    EditDistDP(str1, str2);
}
}

// This code is contributed by aashish1995
```

Output

```
4
```

Time Complexity: O(m x n)
Auxiliary Space: O(m)

This is a memoized version of recursion i.e. Top-Down DP:

we are more focussed in ~~bottom up~~ top down no recursions here wahi.

C++14 | Java | Python3 | C# | Javascript

```
import java.util.*;
class GFG {

    static int minDis(String s1, String s2, int n, int m,
                      int[][] dp)
    {

        // If any String is empty,
        // return the remaining characters of other String
        if (n == 0)
            return m;
        if (m == 0)
            return n;

        // To check if the recursive tree
        // for given n & m has already been executed
        if (dp[n][m] != -1)
            return dp[n][m];

        // If characters are equal, execute
        // recursive function for n-1, m-1
        if (s1.charAt(n - 1) == s2.charAt(m - 1)) {
            return dp[n][m] = minDis(s1, s2, n - 1, m - 1, dp);
        }
        // If characters are nt equal, we need to
        // find the minimum cost out of all 3 operations.
        else {

            int insert, del, replace; // temp variables

            insert = minDis(s1, s2, n, m - 1, dp);
```

```
            del = minDis(s1, s2, n - 1, m, dp);
            replace = minDis(s1, s2, n - 1, m - 1, dp);

            return dp[n][m]
                = 1 + Math.min(insert, Math.min(del, replace));
        }
    }
    // Driver program
    public static void main(String[] args)
    {

        String str1 = "voldemort";
        String str2 = "dumbledore";

        int n = str1.length(), m = str2.length();
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 0; i < n + 1; i++)
            Arrays.fill(dp[i], -1);
        System.out.print(minDis(str1, str2, n, m, dp));
    }
}

// This code is contributed by gauravrajput1
```

Output

```
7
```

Time Complexity: O(m x n)
Auxiliary Space: O(m *n)+O(m+n)

(m*n) extra array space and (m+n) recursive stack space.

Applications: There are many practical applications of edit distance algorithm, refer Lucene API for sample. Another example, display all the words in a dictionary that are near proximity to a given wordincorrectly spelled word.

https://youtu.be/Thv3TfsZVpw

Thanks to Vivek Kumar for suggesting updates.
Thanks to **Venki** for providing initial post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Words: 606
Characters: 3610

Post Title

Longest Increasing Subsequence | DP-3

We have already discussed Overlapping Subproblems and Optimal Substructure properties. Now, let us discuss the Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.

arr[ ]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

Examples:

```
Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3
The longest increasing subsequence is 3, 10, 20

Input: arr[] = {3, 2}
Output: Length of LIS = 1
The longest increasing subsequences are {3} and {2}

Input: arr[] = {50, 3, 10, 7, 40, 80}
Output: Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.Div: practiceLinkDiv

Div: extraAd

Method 1: Recursion.
Optimal Substructure: Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

Then, L(i) can be recursively written as:

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

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.
Formally, the length of the longest increasing subsequence ending at index i, will be 1 greater than the maximum of lengths of all longest increasing subsequences ending at indices before i, where arr[j] < arr[i] (j < i).

Write

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

The recursive tree given below will make the approach clearer:

```
Input  : arr[] = {3, 10, 2, 11}
f(i): Denotes LIS of subarray ending at index 'i'

(LIS(1)=1)

        f(4)  {f(4) = 1 + max(f(1), f(2), f(3))}
  /      |    \
f(1)    f(2)  f(3) {f(3) = 1, f(2) and f(1) are > f(3)}
        |       |  \
        f(1)    f(2)  f(1) {f(2) = 1 + max(f(1)}
                |
                f(1) {f(1) = 1}
```

Below is the implementation of the recursive approach:

C++ C Java Python3 C# Javascript

```
/* A Naive Java Program for LIS Implementation */
class LIS {
    static int max_ref; // stores the LIS

    /* To make use of recursive calls, this function must
    return two things: 1) Length of LIS ending with element
    arr[n-1]. We use max_ending_here for this purpose 2)
    Overall maximum as the LIS may end with an element
       before arr[n-1] max_ref is used this purpose.
    The value of LIS of full array of size n is stored in
    *max_ref which is our final result */
    static int _lis(int arr[], int n)
    {
        // base case
        if (n == 1)
            return 1;

        // 'max_ending_here' is length of LIS ending with
        // arr[n-1]
        int res, max_ending_here = 1;

        /* Recursively get all LIS ending with arr[0],
           arr[1] ... arr[n-2]. If   arr[i-1] is smaller
           than arr[n-1], and max ending with arr[n-1] needs
           to be updated, then update it */
        for (int i = 1; i < n; i++) {
            res = _lis(arr, i);
            if (arr[i - 1] < arr[n - 1]
```

```
                && res + 1 > max_ending_here)
                max_ending_here = res + 1;

        }

        // Compare max_ending_here with the overall max. And
        // update the overall max if needed
        if (max_ref < max_ending_here)
            max_ref = max_ending_here;

        // Return length of LIS ending with arr[n-1]
        return max_ending_here;
    }

    // The wrapper function for _lis()
    static int lis(int arr[], int n)
    {
        // The max variable holds the result
        max_ref = 1;

        // The function _lis() stores its result in max
        _lis(arr, n);

        // returns max
        return max_ref;
    }

    // driver program to test above functions
    public static void main(String args[])
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.length;
        System.out.println("Length of lis is " + lis(arr, n)
                           + "\n");
    }
}
/*This code is contributed by Rajat Mishra*/
```

Output

```
Length of lis is 5
```

Complexity Analysis:

- **Time Complexity:** The time complexity of this recursive approach is exponential as there is a case of overlapping subproblems as explained in the recursive tree diagram above.
- **Auxiliary Space:** O(1). No external space used for storing values apart from the internal stack space.

Method 2: Dynamic Programming.
We can see that there are many subproblems in the above recursive solution which are solved

again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation.

The simulation of approach will make things clear:

```
Input   : arr[] = {3, 10, 2, 11}
LIS[] = {1, 1, 1, 1} (initially)
```

Iteration-wise simulation :

1. arr[2] > arr[1] {LIS[2] = max(LIS [2], LIS[1]+1)=2}
2. arr[3] < arr[1] {No change}
3. arr[3] < arr[2] {No change}
4. arr[4] > arr[1] {LIS[4] = max(LIS [4], LIS[1]+1)=2}
5. arr[4] > arr[2] {LIS[4] = max(LIS [4], LIS[2]+1)=3}
6. arr[4] > arr[3] {LIS[4] = max(LIS [4], LIS[3]+1)=3}

We can avoid recomputation of subproblems by using tabulation as shown in the below code:

Below is the implementation of the above approach:

C++ | Java | Python3 | C# | Javascript

```
/* Dynamic Programming Java implementation
   of LIS problem */

class LIS {
    /* lis() returns the length of the longest
       increasing subsequence in arr[] of size n */
    static int lis(int arr[], int n)
    {
        int lis[] = new int[n];
        int i, j, max = 0;

        /* Initialize LIS values for all indexes */
        for (i = 0; i < n; i++)
            lis[i] = 1;

        /* Compute optimized LIS values in
           bottom up manner */
        for (i = 1; i < n; i++)
            for (j = 0; j < i; j++)
                if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;

        /* Pick maximum of all LIS values */
        for (i = 0; i < n; i++)
            if (max < lis[i])
                max = lis[i];

        return max;
```

```
    }

    public static void main(String args[])
    {
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.length;
        System.out.println("Length of lis is " + lis(arr, n)
                           + "\n");
    }
}
/*This code is contributed by Rajat Mishra*/
```

Output

```
Length of lis is 5
```

Complexity Analysis:

- **Time Complexity:** O(n^2).
  As nested loop is used.
- **Auxiliary Space:** O(n).
  Use of any array to store LIS values at each index.

Note: The time complexity of the above Dynamic Programming (DP) solution is O(n^2) and there is a O(N log N) solution for the LIS problem. We have not discussed the O(N log N) solution here as the purpose of this post is to explain Dynamic Programming with a simple example. See below post for O(N log N) solution.
Longest Increasing Subsequence Size (N log N)

Method 3: Dynamic Programming

If we closely observe the problem then we can convert this problem to longest Common Subsequence Problem. Firstly we will create another array of unique elements of original array and sort it. Now the longest increasing subsequence of our array must be present as a subsequence in our sorted array. That's why our problem is now reduced to finding the common subsequence between the two arrays.

```
Eg. arr =[50,3,10,7,40,80]
    // Sorted array
    arr1 = [3,7,10,40,50,80]
    // LIS is longest common subsequence between the two arrays
    ans = 4
    The longest increasing subsequence is {3, 7, 40, 80}
```

C++ | Java | Python3

```
import static java.lang.Math.max;

import java.util.SortedSet;
import java.util.TreeSet;
```

```
import java.util.TreeSet;

// Dynamic Programming Approach of Finding LIS by reducing
// the problem to longest common Subsequence
public class Main {

    /* lis() returns the length of the longest
    increasing subsequence in arr[] of size n */
    static int lis(int arr[], int n)
    {
        SortedSet<Integer> hs = new TreeSet<Integer>();
        // Storing and Sorting unique elements.
        for (int i = 0; i < n; i++)
            hs.add(arr[i]);
        int lis[] = new int[hs.size()];
        int k = 0;
        // Storing all the unique values in a sorted manner.
        for (int val : hs) {
            lis[k] = val;
            k++;
        }
        int m = k, i, j;
        int dp[][] = new int[m + 1][n + 1];

        // Storing -1 in dp multidimensional array.
        for (i = 0; i < m + 1; i++) {
            for (j = 0; j < n + 1; j++) {
                dp[i][j] = -1;
            }
        }

        // Finding the Longest Common Subsequence of the two
        // arrays
        for (i = 0; i < m + 1; i++) {
            for (j = 0; j < n + 1; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                }
                else if (arr[j - 1] == lis[i - 1]) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                }
                else {
                    dp[i][j]
                        = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
    }

    // Driver Program for the above test function.
    public static void main(String[] args)
    {
```

(This will take O(n2) still

```
        int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
        int n = arr.length;

        System.out.println("Length of lis is " + lis(arr, n)
                           + "\n");
    }
}

// This Code is Contributed by Omkar Subhash Ghongade
```

Output

```
Length of lis is 5
```

Complexity Analysis : O(n*n)

As nested loop is used

Space Complexity : O(n*n)

As a matrix is used for storing the values.

Method 4 : Memoization DP

This is extension of recursive method

We can see that there are many subproblems in the above recursive solution which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization

C++ | Java | Python3

```
/* A Memoization Java Program for LIS Implementation */
import java.util.Arrays;
import java.lang.*;
class LIS {

  /* To make use of recursive calls, this function must
     return two things: 1) Length of LIS ending with element
     arr[n-1]. We use max_ending_here for this purpose 2)
     Overall maximum as the LIS may end with an element
        before arr[n-1] max_ref is used this purpose.
     The value of LIS of full array of size n is stored in
     *max_ref which is our final result */
  static int f(int idx, int prev_idx, int n, int a[],
               int[][] dp)
  {
    if (idx == n) {
      return 0;
    }

    if (dp[idx][prev_idx + 1] != -1) {
      return dp[idx][prev_idx + 1];
```

```
    }

    int notTake = 0 + f(idx + 1, prev_idx, n, a, dp);
    int take = Integer.MIN_VALUE;
    if (prev_idx == -1 || a[idx] > a[prev_idx]) {
      take = 1 + f(idx + 1, idx, n, a, dp);
    }

    return dp[idx][prev_idx + 1] = Math.max(take, notTake);
  }

  // The wrapper function for _lis()
  static int lis(int arr[], int n)
  {

    // The function _lis() stores its result in max
    int dp[][] = new int[n+1][n+1];
    for (int row[] : dp)
      Arrays.fill(row, -1);

    return f(0, -1, n, arr, dp);


  }

  // driver program to test above functions
  public static void main(String args[])
  {
    int a[] = { 3, 10, 2, 1, 20 };
    int n = a.length;
    System.out.println("Length of lis is " + lis(a, n)
                       + "\n");
  }
}

// This code is contributed by Sanskar.
```

Output

```
Length of lis is 3
```

Complexity Analysis:

Time Complexity: $O(n^2)$.
Auxiliary Space: $O(n^2)$.

- Printing LIS of array
- Recent articles based on LIS!

Post Title

Longest Common Substring | DP-29

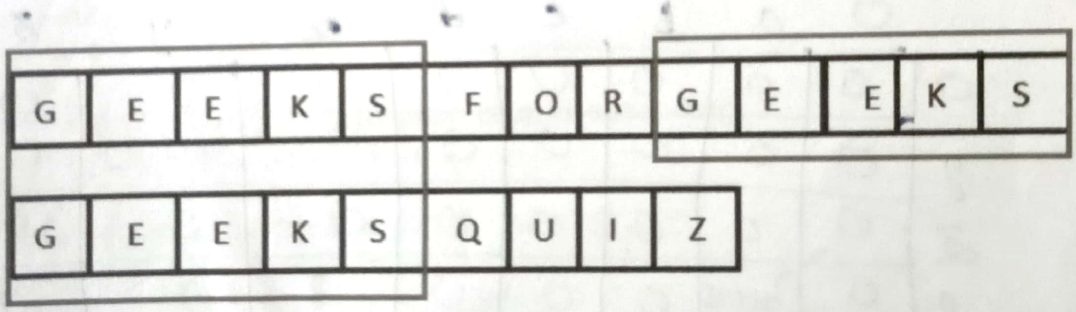
Given two strings 'X' and 'Y', find the length of the longest common substring.

Examples :

Input : *X = "GeeksforGeeks", y = "GeeksQuiz"*
Output : *5*
Explanation:
The longest common substring is "Geeks" and is of length 5.

Input : *X = "abcdxyz", y = "xyzabcd"*
Output : *4*
Explanation:
The longest common substring is "abcd" and is of length 4.

Input : *X = "zxabcdezy", y = "yzabcdezx"*
Output : *6*
Explanation:
The longest common substring is "abcdez" and is of length 6.



OUTPUT: 5
As longest Common String is "Geeks"

Recommended Practice
Longest Common Substring
Try It!
Div: practiceLinkDiv

Approach:
Let m and n be the lengths of the first and second strings respectively.

A **simple solution** is to one by one consider all substrings of the first string and for every substring check if it is a substring in the second string. Keep track of the maximum length substring. There

When Match → add 1 in Previous diagonal
when not match = 0

	0	G	E	E	K	S
0	0	0	0	0	0	0
G	0	0+1=1 (Match)	0	0	0	0
E	0	0	1+1=2	0+1=1	0	0
E	0	0	0+1=1	2+1=3	0	0
K	0	0	0	0	3+1=4	0
S	0	0	0	0	0	4+1=5
Q	0	0	Not Match	0	0	0
U	0	0	0	0	0	0
I	0	0	0	0	0	0
Z	0	0	0	0	0	0

→ Max = result = 5
since it's substring so continuous diagonal
Unlike Subsequence where we store previous also indeed 0's

	0	a	b	c	d	e	y	z
0	0	0	0	0	0	0	0	0
y	0	0	0	0	0	0	0+1=1	0
z	0	0	0	0	0	0	0	1+1=2
a	0	0+1=1	0	0	0	0	0	0
b	0	0	1+1=2	0	0	0	0	0
c	0	0	0	2+1=3	0	0	0	0
d	0	0	0	0	3+1=4	0	0	0
e	0	0	0	0	0	0	0	0
z	0	0	0	0	0	0	0	0+1=1
a	0	0	0	0	0	0+1=1	0	0

Max

will be O(m^2) substrings and we can find whether a string is substring on another string in O(n) time (See this). So overall time complexity of this method would be $O(n * m^2)$

Dynamic Programming can be used to find the longest common substring in O(m*n) time. The idea is to find the length of the longest common suffix for all substrings of both strings and store these lengths in a table.

The longest common suffix has following optimal substructure property.
If last characters match, then we reduce both lengths by 1
LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 if X[m-1] = Y[n-1]
If last characters do not match, then result is 0, i.e.,
LCSuff(X, Y, m, n) = 0 if (X[m-1] != Y[n-1])
Now we consider suffixes of different substrings ending at different indexes.
The maximum length Longest Common Suffix is the longest common substring.
LCSubStr(X, Y, m, n) = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m and 1 <= j <= n

Following is the iterative implementation of the above solution.

C++ | Java | Python3 | C# | PHP | Javascript

```
//  Java implementation of
// finding length of longest
// Common substring using
// Dynamic Programming
class GFG {
    /*
       Returns length of longest common substring
       of X[0..m-1] and Y[0..n-1]
    */
    static int LCSubStr(char X[], char Y[],
                        int m, int n)
    {
        // Create a table to store
        // lengths of longest common
        // suffixes of substrings.
        // Note that LCSuff[i][j]
        // contains length of longest
        // common suffix of
        // X[0..i-1] and Y[0..j-1].
        // The first row and first
        // column entries have no
        // logical meaning, they are
        // used only for simplicity of program
        int LCStuff[][] = new int[m + 1][n + 1];

        // To store length of the longest
        // common substring
        int result = 0;
```

```
    // Following steps build
    // LCSuff[m+1][n+1] in bottom up fashion
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                LCStuff[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
            {
                LCStuff[i][j]
                    = LCStuff[i - 1][j - 1] + 1;
                result = Integer.max(result,
                                     LCStuff[i][j]);
            }
            else
                LCStuff[i][j] = 0;
        }
    }
    return result;
}

// Driver Code
public static void main(String[] args)
{
    String X = "OldSite:GeeksforGeeks.org";
    String Y = "NewSite:GeeksQuiz.com";

    int m = X.length();
    int n = Y.length();

    System.out.println(LCSubStr(X.toCharArray(),
                                Y.toCharArray(), m,
                    n));
}
}

// This code is contributed by Sumit Ghosh
```

Output

```
Length of Longest Common Substring is 10
```

Time Complexity: O(m*n)
Auxiliary Space: O(m*n), since m*n extra space has been taken.

Another approach: (Space optimized approach).
In the above approach, we are only using the last row of the 2-D array only, hence we can optimize the space by using
a 2-D array of dimension 2*(min(n,m)).

Below is the implementation of the above approach:

C++ | Java | Python3 | C# | Javascript

```
// Java implementation of the above approach

class GFG
{

    // Function to find the length of the
    // longest LCS
    static int LCSubStr(String s,String t,
                        int n,int m)
    {

        // Create DP table
        int dp[][]=new int[2][m+1];
        int res=0;

        for(int i=1;i<=n;i++)
        {
            for(int j=1;j<=m;j++)
            {
                if(s.charAt(i-1)==t.charAt(j-1))
                {
                    dp[i%2][j]=dp[(i-1)%2][j-1]+1;
                    if(dp[i%2][j]>res)
                        res=dp[i%2][j];
                }
                else dp[i%2][j]=0;
            }
        }
        return res;
    }

    // Driver Code
    public static void main (String[] args)
    {
        String X="OldSite:GeeksforGeeks.org";
        String Y="NewSite:GeeksQuiz.com";

        int m=X.length();
        int n=Y.length();

        // Function call
        System.out.println(LCSubStr(X,Y,m,n));

    }
}
```

Output

```
10
```

Time Complexity: O(n*m)
Auxiliary Space: O(min(m,n))

Another approach: (Using recursion)
Here is the recursive solution of the above approach.

C++ | Java | Python3 | C# | PHP | Javascript

```
// Java program using to find length of the
// longest common substring recursion

class GFG {

    static String X, Y;
    // Returns length of function
    // for longest common
    // substring of X[0..m-1] and Y[0..n-1]
    static int lcs(int i, int j, int count)
    {

        if (i == 0 || j == 0)
        {
            return count;
        }

        if (X.charAt(i - 1)
            == Y.charAt(j - 1))
        {
            count = lcs(i - 1, j - 1, count + 1);
        }
        count = Math.max(count,
                         Math.max(lcs(i, j - 1, 0),
                                  lcs(i - 1, j, 0)));
        return count;
    }

    // Driver code
    public static void main(String[] args)
    {
        int n, m;
        X = "abcdxyz";
        Y = "xyzabcd";

        n = X.length();
        m = Y.length();

        System.out.println(lcs(n, m, 0));
    }
```

```
}
// This code is contributed by Rajput-JI
```

Output

```
4
```

Maximum Space Optimization:

1. In this method, we will use recursion to find the longest prefix of all the possible substrings.
2. Let LongestCommonSuffix\left(i, j\right) gives the length of the longest common suffix starting from indices *i*, *j* of strings *X*, *Y* respectively.
3. Then the function can be defined as : LongestCommonSuffix\left(i, j\right) = \begin{cases} 1 + LongestCommonSuffix\left(i + 1, j + 1\right) & \text{if} & 0 \le i \lt \left|X\right| & \text{and} & 0 \le j \lt \left|Y\right| & \text{and} & X\left[i\right] = Y\left[j\right] \\ 0 & \text{otherwise} \end{cases}
4. In this recursion we can see that the function has only one dependency, so that means we can get away with memorizing just the previous computation if we do our computation in a specific order.
5. Consider the following table where we memorize the solutions:

	0	1	2	3	4
0					
1					
2					
3					

We need to find the solution diagonally upwards. In this particular example:

- first diagonal
  - (4, 0)
- second diagonal
  - (4, 1)
  - (3, 0)
- third diagonal
  - (4, 2)
  - (3, 1)
  - (2, 0)
- ...

Like this, we need to remember only the previous computation.

Python3

```
# Python code for the above approach
from functools import lru_cache
from operator import itemgetter
```

```
    # function to find the longest common substring

    # Memorizing with maximum size of the memory as 1
    @lru_cache(maxsize=1)

    # function to find the longest common prefix
    def longest_common_prefix(i: int, j: int) -> int:

        if 0 <= i < len(x) and 0 <= j < len(y) and x[i] == y[j]:
            return 1 + longest_common_prefix(i + 1, j + 1)
        else:
            return 0

    # diagonally computing the subproblems
    # to decrease memory dependency
    def digonal_computation():

        # upper right triangle of the 2D array
        for k in range(len(x)):
            yield from ((longest_common_prefix(i, j), i, j)
                        for i, j in zip(range(k, -1, -1),
                                        range(len(y) - 1, -1, -1)))

        # lower left triangle of the 2D array
        for k in range(len(y)):
            yield from ((longest_common_prefix(i, j), i, j)
                        for i, j in zip(range(k, -1, -1),
                                        range(len(x) - 1, -1, -1)))

    # returning the maximum of all the subproblems
    return max(digonal_computation(), key=itemgetter(0), default=(0, 0, 0

# Driver Code
if __name__ == '__main__':
    x: str = 'GeeksforGeeks'
    y: str = 'GeeksQuiz'
    length, i, j = longest_common_substring(x, y)
    print(f'length: {length}, i: {i}, j: {j}')
    print(f'x substring: {x[i: i + length]}')
    print(f'y substring: {y[j: j + length]}')
```

Output

```
length: 5, i: 0, j: 0
x substring: Geeks
y substring: Geeks
```

Write

08/11/2022, 12:36

Time Complexity: O\left(\left|X\right|\left|Y\right|\right)
Space Complexity: O\left(1\right)

Exercise: The above solution prints only the length of the longest common substring. Extend the solution to print the substring also.

Words: 594
Characters: 3694

(~~Find~~ Store the index & max.
Just diagonally go previous until '0'
you will get your substring.

Post Title

Subset Sum Problem | DP-25

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

Example:

```
Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9
Output: True
There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: False
There is no subset that add up to 30.
```

Method 1: Recursion.
Approach: For the recursive approach we will consider two cases.

1. Consider the last element and now the **required sum = target sum - value of 'last' element** and **number of elements = total elements - 1**
2. Leave the 'last' element and now the **required sum = target sum** and **number of elements = total elements - 1**

Following is the recursive formula for isSubsetSum() problem.

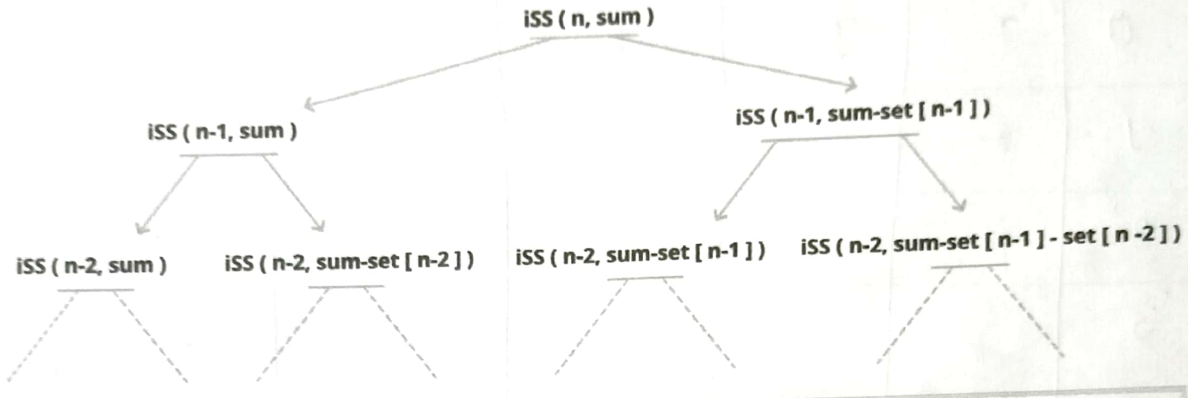
```
isSubsetSum(set, n, sum)
= isSubsetSum(set, n-1, sum) ||
  isSubsetSum(set, n-1, sum-set[n-1])
Base Cases:
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
isSubsetSum(set, n, sum) = true, if sum == 0
```

Let's take a look at the simulation of above approach-:

```
set[]={3, 4, 5, 2}
sum=9
(x, y)= 'x' is the left number of elements,
'y' is the required sum

                (4, 9)
               {True}
              /      \
         (3, 6)       (3, 9)

        /    \        /   \
```

```
      (2, 2)  (2, 6)   (2, 5)  (2, 9)
      {True}
      /   \
   (1, -3) (1, 2)
{False}  {True}
          /    \
        (0, 0)  (0, 2)
        {True} {False}
```

ISS = isSubsetSum



C++ C Java Python3 C# PHP Javascript

```
// A recursive solution for subset sum
// problem
class GFG {

    // Returns true if there is a subset
    // of set[] with sum equal to given sum
    static boolean isSubsetSum(int set[],
                               int n, int sum)
    {
        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0)
            return false;

        // If last element is greater than
        // sum, then ignore it
        if (set[n - 1] > sum)
            return isSubsetSum(set, n - 1, sum);

        /* else, check if sum can be obtained
        by any of the following
         (a) including the last element
         (b) excluding the last element */
        return isSubsetSum(set, n - 1, sum)
            || isSubsetSum(set, n - 1, sum - set[n - 1]);
    }
```

Let's better put 'i' as target sum

& 'j' as subsets :-

→ subset(j)

X	0	3	4	5	2
0	T	T	T	T	T
1	F				
2	F				
3	F				
4	F				
5	F				
6	F				

↓ Target sum (i)

Now iterate for (i = 1 to ~~targetSum~~)

for (j = 1st element to last element taken together)

if (i < set[j-1])

//means let's not pick that value

so DP[i][j] = DP[i-1][j] // Take previous case answer

}

else {

DP[i][j] = DP[i-1][j] ||

DP[i-1][j-

```
    /* Driver code */
    public static void main(String args[])
    {
        int set[] = { 3, 34, 4, 12, 5, 2 };
        int sum = 9;
        int n = set.length;
        if (isSubsetSum(set, n, sum) == true)
            System.out.println("Found a subset"
                               + " with given sum");

        else
            System.out.println("No subset with"
                               + " given sum");

    }
}

/* This code is contributed by Rajat Mishra */
```

Output

```
Found a subset with given sum
```

Complexity Analysis: The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (*There is no known polynomial time solution for this problem*).

Method 2: To solve the problem in Pseudo-polynomial time use the Dynamic programming. So we will create a 2D array of size (arr.size() + 1) * (target + 1) of type **boolean**. The state DP[i][j] will be **true** if there exists a subset of elements from A[0....i] with **sum value = 'j'.** The approach for the problem is:

```
if (A[i-1] > j)
DP[i][j] = DP[i-1][j]
else
DP[i][j] = DP[i-1][j] OR DP[i-1][j-A[i-1]]
```

1. This means that if current element has value greater than 'current sum value' we will copy the answer for previous cases
2. And if the current sum value is greater than the 'ith' element we will see if any of previous states have already experienced the **sum='j' OR any previous states experienced a value 'j - A[i]'** which will solve our purpose.

The below simulation will clarify the above approach:

```
set[]={3, 4, 5, 2}
target=6

    0   1   2   3   4   5   6
0   T   F   F   F   F   F   F
```

3	T	F	F	T	F	F	F
4	T	F	F	T	T	F	F
5	T	F	F	T	T	T	F
2	T	F	T	T	T	T	T

Below is the implementation of the above approach:

C C++ Java Python3 C# PHP Javascript

```
// A Dynamic Programming solution for subset
// sum problem
class GFG {

    // Returns true if there is a subset of
    // set[] with sum equal to given sum
    static boolean isSubsetSum(int set[],
                               int n, int sum)
    {
        // The value of subset[i][j] will be
        // true if there is a subset of
        // set[0..j-1] with sum equal to i
        boolean subset[][] = new boolean[sum + 1][n + 1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++)
            subset[0][i] = true;

        // If sum is not 0 and set is empty,
        // then answer is false
        for (int i = 1; i <= sum; i++)
            subset[i][0] = false;

        // Fill the subset table in bottom
        // up manner
        for (int i = 1; i <= sum; i++) {
            for (int j = 1; j <= n; j++) {
                subset[i][j] = subset[i][j - 1];
                if (i >= set[j - 1])
                    subset[i][j] = subset[i][j]
                                   || subset[i - set[j - 1]][j - 1];
            }
        }

        /* // uncomment this code to print table
        for (int i = 0; i <= sum; i++)
        {
        for (int j = 0; j <= n; j++)
```

```
            System.out.println (subset[i][j]);
        } */


        return subset[sum][n];
    }

    /* Driver code*/
    public static void main(String args[])
    {
        int set[] = { 3, 34, 4, 12, 5, 2 };
        int sum = 9;
        int n = set.length;
        if (isSubsetSum(set, n, sum) == true)
            System.out.println("Found a subset"
                               + " with given sum");
        else
            System.out.println("No subset with"
                               + " given sum");
    }
}

/* This code is contributed by Rajat Mishra */
```

Output

```
Found a subset with given sum
```

Complexity Analysis:

Time Complexity: O(sum*n), where sum is the 'target sum' and 'n' is the size of array.
Auxiliary Space: O(sum*n), as the size of 2-D array is sum*n. + O(n) for recursive stack space

Memoization Technique for finding Subset Sum:

Method:

1. In this method, we also follow the recursive approach but In this method, we use another 2-D matrix in we first initialize with -1 or any negative value.
2. In this method, we avoid the few of the recursive call which is repeated itself that's why we use 2-D matrix. In this matrix we store the value of the previous call value.

Below is the implementation of the above approach:

C++ | Java | Python3 | C# | Javascript

```
// Java program for the above approach
class GFG {

    // Check if possible subset with
    // given sum is possible or not
    static int subsetSum(int a[], int n, int sum)
    {
```

```java
package com.company;// The longest common subsequence in Java

class SubSetSum {

    /** Very well explained here the whole concepts of how subset sum code
approach is used */
    public static boolean isSubsetSum(int[] arr, int n, int sum){

        boolean[][] subset = new boolean[n+1][sum+1];

        for (int i=0;i<=n;i++){
            for(int j=0;j<=sum;j++){
                if(i==0 && j==0){  //When we have neither element and sum we want
is 0 as well so always true;
                    subset[i][j]=true;
                    continue;
                }
                if(i==0){  //When we have no element in set
                    subset[i][j]=false;
                    continue;
                }
                if(j==0){  //When we need sum = 0;
                    subset[i][j]=true;
                    continue;
                }

                if(j<arr[i-1]){  //Since we are starting from i =1 to look for
array elements so we use arr[i-1] to take ith element
                    subset[i][j]=subset[i-1][j];  /** This means if current
element has value greater than 'current sum value' we will copy the answer for
previous cases*/
                }
                else{
                    subset[i][j]=subset[i-1][j] || subset[i-1][j-arr[i-1]];  /**
And if the current sume value is greather than the ith element we will see if any
of previous states
                     have already experienced the sum='j' OR any previous states
experience a value 'j-Arr[i]' which will solve our purpose.**/
                }
            }
        }
        for(int i = 0;i<=n;i++){
            for(int j=0;j<=sum;j++){
                System.out.print(subset[i][j]+ " ");
            }
            System.out.println();
        }

        return subset[n][sum];
    }


    public static void main(String[] args) {
        int[] set = {3,4,5,2};
        int sum =6;
        boolean ans = isSubsetSum(set, set.length,sum);
        System.out.println(ans);

    }
}
```

i-1 is our previous state array which help we take as our subproblem

```
    }

    // Storing the value -1 to the matrix
    int tab[][] = new int[n + 1][sum + 1];
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            tab[i][j] = -1;
        }
    }

    // If the sum is zero it means
    // we got our expected sum
    if (sum == 0)
        return 1;

    if (n <= 0)
        return 0;

    // If the value is not -1 it means it
    // already call the function
    // with the same value.
    // it will save our from the repetition.
    if (tab[n - 1][sum] != -1)
        return tab[n - 1][sum];

    // if the value of a[n-1] is
    // greater than the sum.
    // we call for the next value
    if (a[n - 1] > sum)
        return tab[n - 1][sum]
            = subsetSum(a, n - 1, sum);
    else {

        // Here we do two calls because we
        // don't know which value is
        // full-fill our criteria
        // that's why we doing two calls
        if (subsetSum(a, n - 1, sum) != 0
            || subsetSum(a, n - 1, sum - a[n - 1])
                   != 0) {
            return tab[n - 1][sum] = 1;
        }
        else
            return tab[n - 1][sum] = 0;
    }
}

  // Driver Code
public static void main(String[] args)
{

    int n = 5;
    int a[] = { 1, 5, 3, 7, 4 };
    int sum = 12;
```

For more we are only looking for tabulation. & No Memoization (Recursion)

```
        if (subsetSum(a, n, sum) != 0) {
            System.out.println("YES\n");
        }
        else
            System.out.println("NO\n");
    }
}

// This code is contributed by rajsanghavi9.
```

Output

```
YES
```

Complexity Analysis:

- **Time Complexity:** O(sum*n), where sum is the 'target sum' and 'n' is the size of array.
- **Auxiliary Space:** O(sum*n) + O(n) -> O(sum*n) = the size of 2-D array is sum*n and O(n)=auxiliary stack space.

Subset Sum Problem in O(sum) space
Perfect Sum Problem (Print all subsets with given sum)
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Words: 649
Characters: 3831

Post Title

Partition problem | DP-18

(It's just application of subset sum DP Approach.

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same.

Examples:

> ***Input:*** *arr[] = {1, 5, 11, 5}*
> ***Output:*** *true*
> *The array can be partitioned as {1, 5, 5} and {11}*
>
> ***Input:*** *arr[] = {1, 5, 3}*
> ***Output:*** *false*
> *The array cannot be partitioned into equal sum sets.*

Following are the two main steps to solve this problem:
1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.
The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

To me it looks like find min coin to sum. Here just coins are not infinite numbers. Can be used only once.

Recursive Solution (No need to do anymore, you can directly start with DP approach only)

Dynamic Programming Solution
1. **Top-Down:** Memoization

We can avoid the repeated work done in method 1 by storing the result calculated so far.

We just need to store all the values in a matrix.

C++ | Java | Python3 | C# | Javascript

```
// Java program for partition problem
import java.io.*;
import java.util.*;

class GFG {

    // A utility function that returns true if there is
    // a subset of arr[] with sum equal to given sum
    static int isSubsetSum(int arr[], int n, int sum,
                           int[][] dp)
    {
        // Base Cases
        if (sum == 0)
```

```
        return 1;
    if (n == 0 && sum != 0)
        return 0;

    // return solved subproblem
    if (dp[n][sum] != -1) {
        return dp[n][sum];
    }

    // If last element is greater than sum, then
    // ignore it
    if (arr[n - 1] > sum)
        return isSubsetSum(arr, n - 1, sum, dp);

    /* else, check if sum can be obtained by any of
          the following
          (a) including the last element
          (b) excluding the last element
    */
    // also store the subproblem in dp matrix
    if (isSubsetSum(arr, n - 1, sum, dp) != 0
        || isSubsetSum(arr, n - 1, sum - arr[n - 1], dp)
               != 0)
        return dp[n][sum] = 1;
    return dp[n][sum] = 0;
    // return dp[n][sum] = isSubsetSum(arr, n - 1, sum,
    // dp) || isSubsetSum(arr, n - 1, sum - arr[n - 1],
    // dp);
}

// Returns true if arr[] can be partitioned in two
// subsets of equal sum, otherwise false
static int findPartiion(int arr[], int n)
{
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // If sum is odd, there cannot be two subsets
    // with equal sum
    if (sum % 2 != 0)
        return 0;

    // To store overlapping subproblems
    int dp[][] = new int[n + 1][sum + 1];
    for (int row[] : dp)
        Arrays.fill(row, -1);

    // Find if there is subset with sum equal to
    // half of total sum
    return isSubsetSum(arr, n, sum / 2, dp);
```

```
    }

    public static void main(String[] args)
    {
        int arr[] = { 3, 1, 5, 9, 12 };
        int n = arr.length;

        // Function call
        if (findPartiion(arr, n) == 1)
            System.out.println(
                "Can be divided into two subsets of equal sum");
        else
            System.out.println(
                "Can not be divided into two subsets of equal sum");

        int arr2[] = { 3, 1, 5, 9, 14 };
        int n2 = arr2.length;

        if (findPartiion(arr2, n2) == 1)
            System.out.println(
                "Can be divided into two subsets of equal sum");
        else
            System.out.println(
                "Can not be divided into two subsets of equal sum");
    }
}
```

Output

```
Can be divided into two subsets of equal sum
Can not be divided into two subsets of equal sum
```

Time Complexity: O(sum*n)

Auxiliary Space: O(sum*n)

2. **Bottom-Up:** Tabulation

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array part[][] of size (sum/2 + 1)*(n+1). And we can construct the solution in a bottom-up manner such that every filled entry has the following property

```
part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum
             equal to i, otherwise false
```

C++ C Java Python3 C# Javascript

```
// A dynamic programming based Java program for partition
// problem
import java.io.*;
```

```
class Partition {

    // Returns true if arr[] can be partitioned in two
    // subsets of equal sum, otherwise false
    static boolean findPartition(int arr[], int n)
    {
        int sum = 0;
        int i, j;

        // Calculate sum of all elements
        for (i = 0; i < n; i++)
            sum += arr[i];

        if (sum % 2 != 0)
            return false;

        boolean part[][] = new boolean[sum / 2 + 1][n + 1];

        // initialize top row as true
        for (i = 0; i <= n; i++)
            part[0][i] = true;

        // initialize leftmost column, except part[0][0], as
        // 0
        for (i = 1; i <= sum / 2; i++)
            part[i][0] = false;

        // Fill the partition table in bottom up manner
        for (i = 1; i <= sum / 2; i++) {
            for (j = 1; j <= n; j++) {
                part[i][j] = part[i][j - 1];
                if (i >= arr[j - 1])
                    part[i][j]
                        = part[i][j]
                          || part[i - arr[j - 1]][j - 1];
            }
        }

        /* // uncomment this part to print table
        for (i = 0; i <= sum/2; i++)
        {
            for (j = 0; j <= n; j++)
                printf ("%4d", part[i][j]);
            printf("\n");
        } */

        return part[sum / 2][n];
    }

    // Driver code
    public static void main(String[] args)
```

F or T → T
T or F → T
F or F → F
T or T → T

```
    {
        int arr[] = { 3, 1, 1, 2, 2, 1 };
        int n = arr.length;
        if (findPartition(arr, n) == true)
            System.out.println(
                "Can be divided into two " "subsets of equal sum");
        else
            System.out.println(
                "Can not be divided into" " two subsets of equal sum");
    }
}
/* This code is contributed by Devesh Agrawal */
```

Output

```
Can be divided into two subsets of equal sum
```

Following diagram shows the values in the partition table.

	{}	{3}	{3,1}	{3,1,1}	{3,1,1,2}	{3,1,1,2,2}	{3,1,1,2,2,1}
0	True	True	True	True	True	True	True
1	False	False	True	True	True	True	True
2	False	False	False	True	True	True	True
3	False	True	True	True	True	True	True
4	False	False	True	True	True	True	True
5	False	False	False	True	True	True	True

Time Complexity: O(sum*n)
Auxiliary Space: O(sum*n)

Please note that this solution will not be feasible for arrays with big sum.

Dynamic Programming Solution (Space Complexity Optimized)

Instead of creating a 2-D array of size (sum/2 + 1)*(n + 1), we can solve this problem using an array of size (sum/2 + 1) only.

> *part[j] = true if there is a subset with sum equal to j, otherwise false.*

Below is the implementation of the above approach:

C++ | Java | Python3 | C# | Javascript

```
// A Dynamic Programming based
```

```
// .. Dynamic Programming based
// Java program to partition problem
import java.io.*;

class GFG{

// Returns true if arr[] can be partitioned
// in two subsets of equal sum, otherwise false
public static boolean findPartiion(int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Calculate sum of all elements
    for(i = 0; i < n; i++)
        sum += arr[i];

    if (sum % 2 != 0)
        return false;

    boolean[] part = new boolean[sum / 2 + 1];

    // Initialize the part array
    // as 0
    for(i = 0; i <= sum / 2; i++)
    {
        part[i] = false;
    }

    // Fill the partition table in
    // bottom up manner
    for(i = 0; i < n; i++)
    {

        // The element to be included
        // in the sum cannot be
        // greater than the sum
        for(j = sum / 2; j >= arr[i]; j--)
        {

            // Check if sum - arr[i] could be
            // formed from a subset using elements
            // before index i
            if (part[j - arr[i]] == true || j == arr[i])
                part[j] = true;
        }
    }
    return part[sum / 2];
}

// Driver code
public static void main(String[] args)
{
```

```
    int arr[] = { 1, 3, 3, 2, 3, 2 };
    int n = 6;

    // Function call
    if (findPartiion(arr, n) == true)
        System.out.println("Can be divided into two " +
                           "subsets of equal sum");
    else
        System.out.println("Can not be divided into " +
                           "two subsets of equal sum");
}
}

// This code is contributed by RohitOberoi
```

Output

```
Can be divided into two subsets of equal sum
```

Time Complexity: O(sum * n)
Auxiliary Space: O(sum)

Please note that this solution will not be feasible for arrays with big sum.
References:
http://en.wikipedia.org/wiki/Partition_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Words: 423
Characters: 2467

Post Title

0-1 Knapsack Problem | DP-10

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Method 1: Recursion by Brute-Force algorithm OR Exhaustive Search.
Approach: A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.
Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from 'n' items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of 'nth' item is greater than 'W', then the nth item cannot be included and **Case 1** is the only possibility.

Below is the implementation of the above approach:

C++ C Java Python C# PHP Javascript

```
/* A Naive recursive implementation
of 0-1 Knapsack problem */
```

```
class Knapsack {

    // A utility function that returns
    // maximum of two integers
    static int max(int a, int b)
    {
      return (a > b) ? a : b;
    }

    // Returns the maximum value that
    // can be put in a knapsack of
    // capacity W
    static int knapSack(int W, int wt[], int val[], int n)
    {
        // Base Case
        if (n == 0 || W == 0)
            return 0;

        // If weight of the nth item is
        // more than Knapsack capacity W,
        // then this item cannot be included
        // in the optimal solution
        if (wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else
            return max(val[n - 1]
                       + knapSack(W - wt[n - 1], wt,
                                  val, n - 1),
                       knapSack(W, wt, val, n - 1));
    }

    // Driver code
    public static void main(String args[])
    {
        int val[] = new int[] { 60, 100, 120 };
        int wt[] = new int[] { 10, 20, 30 };
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

Value & previously ~~Rest with row~~ that taken value has to be added.

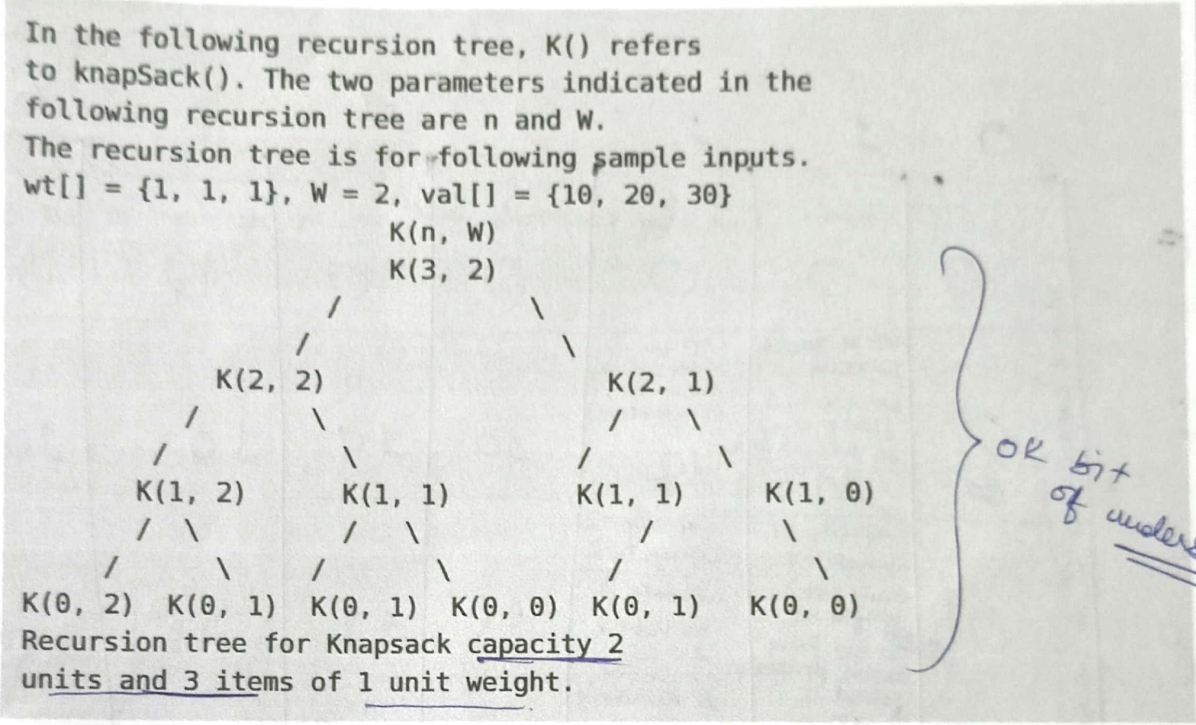
Eg: Taking the last Value.

Ignoring the last value

Output

```
220
```

It should be noted that the above function computes the same sub-problems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. The time complexity of this naive recursive solution is exponential (2^n).



Complexity Analysis:

- **Time Complexity:** O(2^n).
  As there are redundant subproblems.
- **Auxiliary Space :**O(1) + O(N).
  As no extra data structure has been used for storing values but O(N) auxiliary stack space(ASS) has been used for recursion stack.

Since subproblems are evaluated again, this problem has Overlapping Sub-problems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem.

Method 2: Like other typical Dynamic Programming(DP) problems, re-computation of same subproblems can be avoided by constructing a temporary array K[][] in bottom-up manner. Following is Dynamic Programming based implementation.

Approach: In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach. In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.
The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'.
Now two possibilities can take place:

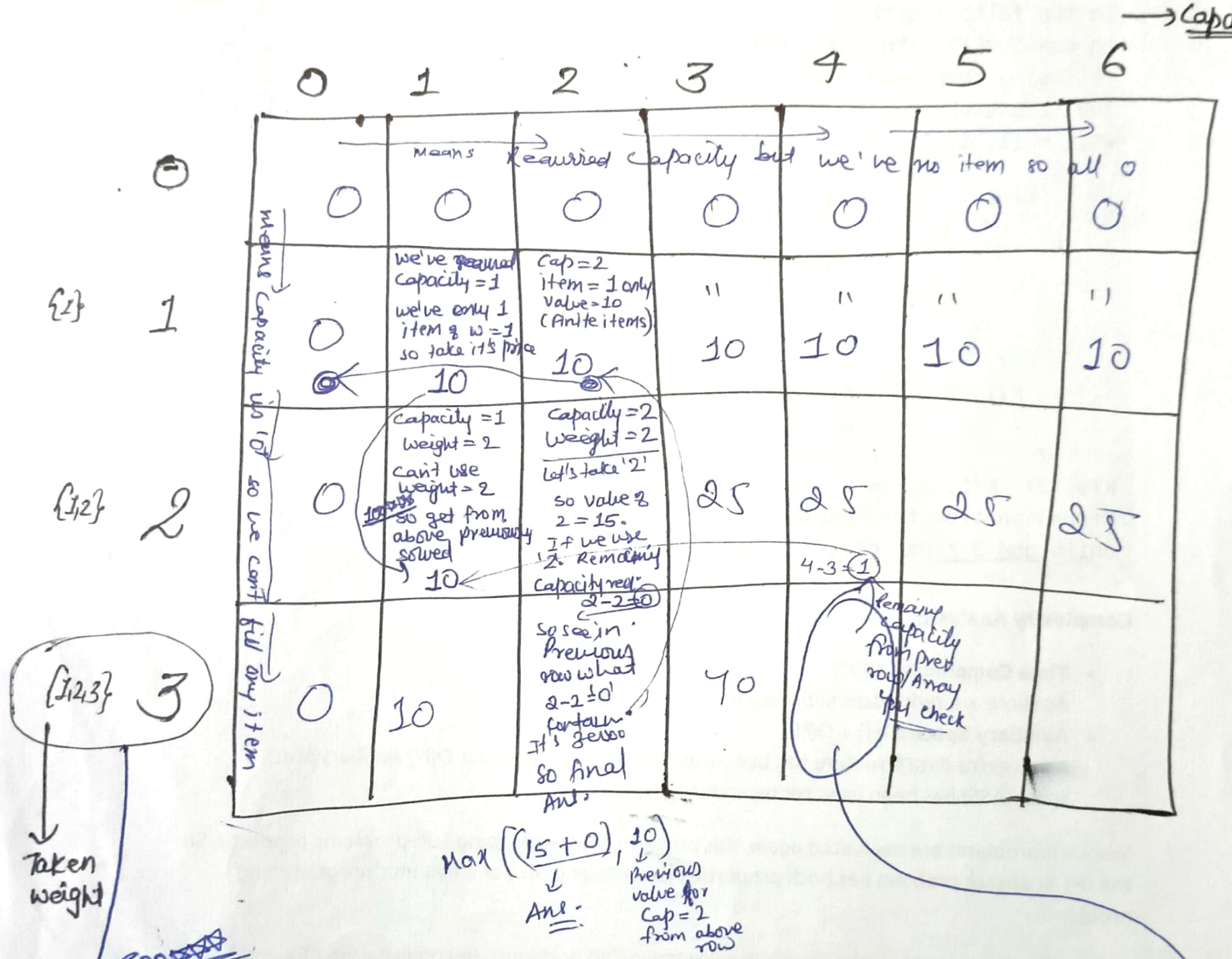
- Fill 'wi' in the given column.
- Do not fill 'wi' in the given column.

Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'wi'+ value of the column weighing 'j-wi' in the previous row. So we take the

1, 2, 3 → weight
10, 15, 40 → value

capacity 6 unit

Assign the items first



Actual meaning of this row:—

Now To put weight '3' taken we've already put '1' & '2' in Previous rows. so overall taken all {1,2,3} items. But deciding only on '3' about how to put it.

we require capacity = 4, we've to fill w = 3, (Rem! 1 & 2 already used)

so since (w = 3) < (capacity 4) → so yes take this

so Value & 3 = 40 → Taken. Now Remaining Capacity = 4 - 3 = 1. Now we want value & best capacity of '1' from previous row which will store best value.

maximum of these two possibilities to fill the current state. This visualisation will make the concept clear:

```
Let weight elements = {1, 2, 3}
Let weight values = {10, 15, 40}
Capacity=6

   0   1   2   3   4   5   6

0  0   0   0   0   0   0   0

1  0  10  10  10  10  10  10

2  0  10  15  25  25  25  25

3  0
```

Explanation:
For filling 'weight = 2' we come across 'j = 3' in which we take maximum of (10, 15 + DP[1][3-2]) = 25
'2' not filled — '2 filled'

```
   0   1   2   3   4   5   6

0  0   0   0   0   0   0   0

1  0  10  10  10  10  10  10

2  0  10  15  25  25  25  25

3  0  10  15  40  50  55  65
```

Explanation:
For filling 'weight=3' we come across 'j=4' in which we take maximum of (25, 40 + DP[2][4-3])
= 50

For filling 'weight=3' we come across 'j=5' in which we take maximum of (25, 40 + DP[2][5-3])
= 55

For filling 'weight=3' we come across 'j=6' in which we take maximum of (25, 40 + DP[2][6-3])
= 65

So that taking the best value

so we got '10'

∴ Capacity value 40+10=50

But also see if in previous row for same capacity if we got best result or not.

so ((40+10), (10)) Max

50

10 ✗✗✗

Max (Val[3] + dp[~~i-1~~ Previous row] [cap-w], dp[Prev Row] [cap])

or

Max (Val[3] + dp[Prev. calc Array] [cap-w], dp[Prev Calc Array] [prev calculat cap value])

Now understood all

C++ C Java Python C# PHP Javascript

```java
// A Dynamic Programming based solution
// for 0-1 Knapsack problem
class Knapsack {

    // A utility function that returns
    // maximum of two integers
    static int max(int a, int b)
    {
          return (a > b) ? a : b;
    }

    // Returns the maximum value that can
    // be put in a knapsack of capacity W
    static int knapSack(int W, int wt[],
                        int val[], int n)
    {
        int i, w;
        int K[][] = new int[n + 1][W + 1];

        // Build table K[][] in bottom up manner
        for (i = 0; i <= n; i++)
        {
            for (w = 0; w <= W; w++)
            {
                if (i == 0 || w == 0)
                    K[i][w] = 0;
                else if (wt[i - 1] <= w)
                    K[i][w]
                        = max(val[i - 1]
                              + K[i - 1][w - wt[i - 1]],
                              K[i - 1][w]);
                else
                    K[i][w] = K[i - 1][w];
            }
        }

        return K[n][W];
    }

    // Driver code
    public static void main(String args[])
    {
        int val[] = new int[] { 60, 100, 120 };
        int wt[] = new int[] { 10, 20, 30 };
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}
```

understood

SOO☆☆☆ and then compare this with previous value. Just in case weight not taken then add max val

If we are including then add that value in our weight minus remaining weight values from previous row.

In case we are not taking that weight. Then just update the previous row value.

SOO☆☆☆ In Knapsack we can't take same item more than one so exclude that here

(unlike rod cutting problem)

```
/*This code is contributed by Rajat Mishra */
```

Output

```
220
```

Complexity Analysis:

- **Time Complexity:** O(N*W).
  where 'N' is the number of weight element and 'W' is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
- **Auxiliary Space:** O(N*W).
  The use of 2-D array of size 'N*W'.

Scope for Improvement :- We used the same approach but with optimized space complexity

C++ | Java | Python3 | C# | Javascript

```
import java.util.*;
class GFG {

  // we can further improve the above Knapsack function's space
  // complexity
  static int knapSack(int W, int wt[], int val[], int n)
  {
    int i, w;
    int [][]K = new int[2][W + 1];

    // We know we are always using the current row or
    // the previous row of the array/vector . Thereby we can
    // improve it further by using a 2D array but with only
    // 2 rows i%2 will be giving the index inside the bounds
    // of 2d array K
    for (i = 0; i <= n; i++) {
      for (w = 0; w <= W; w++) {
        if (i == 0 || w == 0)
          K[i % 2][w] = 0;
        else if (wt[i - 1] <= w)
          K[i % 2][w] = Math.max(
          val[i - 1]
          + K[(i - 1) % 2][w - wt[i - 1]],
          K[(i - 1) % 2][w]);
        else
          K[i % 2][w] = K[(i - 1) % 2][w];
      }
    }
    return K[n % 2][W];
  }

  // Driver Code
```

```
public static void main(String[] args)
{

    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = val.length;

    System.out.print(knapSack(W, wt, val, n));

  }
}

// This code is contributed by gauravrajput1
```

Complexity Analysis:

- **Time Complexity:** O(N*W).
- **Auxiliary Space:** O(2*W)
  As we are using a 2-D array but with only 2 rows.

Method 3: This method uses Memoization Technique (an extension of recursive approach). This method is basically an extension to the recursive approach so that we can overcome the problem of calculating redundant cases and thus increased complexity. We can solve this problem by simply creating a 2-D array that can store a particular state (n, w) if we get it the first time. Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time. This method gives an edge over the recursive approach in this aspect.

C++ | Java | Python3 | C# | Javascript

```
// Here is the top-down approach of
// dynamic programming
class GFG{

// A utility function that returns
// maximum of two integers
static int max(int a, int b)
{
    return (a > b) ? a : b;
}

// Returns the value of maximum profit
static int knapSackRec(int W, int wt[],
                       int val[], int n,
                       int [][]dp)
{

    // Base condition
    if (n == 0 || W == 0)
        return 0;

    if (dp[n][W] != -1)
```

```
        return dp[n][W];

    if (wt[n - 1] > W)

        // Store the value of function call
        // stack in table before return
        return dp[n][W] = knapSackRec(W, wt, val,
                                      n - 1, dp);

    else

        // Return value of table after storing
        return dp[n][W] = max((val[n - 1] +
                               knapSackRec(W - wt[n - 1], wt,
                                           val, n - 1, dp)),
                               knapSackRec(W, wt, val,
                                           n - 1, dp));
}

static int knapSack(int W, int wt[], int val[], int N)
{

    // Declare the table dynamically
    int dp[][] = new int[N + 1][W + 1];

    // Loop to initially filled the
    // table with -1
    for(int i = 0; i < N + 1; i++)
        for(int j = 0; j < W + 1; j++)
            dp[i][j] = -1;

    return knapSackRec(W, wt, val, N, dp);
}

// Driver Code
public static void main(String [] args)
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };

    int W = 50;
    int N = val.length;

    System.out.println(knapSack(W, wt, val, N));
}
}

// This Code is contributed By FARAZ AHMAD
```

Output

220

Complexity Analysis:

- **Time Complexity:** O(N*W).
  As redundant calculations of states are avoided.
- **Auxiliary Space:** O(N*W) + O(N).
  The use of 2D array data structure for storing intermediate states and O(N) auxiliary stack space(ASS) has been used for recursion stack:

[Note: For 32bit integer use long instead of int.]

References:

- http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf
- http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf
- https://youtu.be/T4bY72lCQac?list=PLqM7alHXFySGMu2CSdW_6d2u1o6WFTIO-

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Method 4 :- Again we use the dynamic programming approach with even more optimized space complexity .

C++ | Java | Python3 | C# | Javascript

```
import java.util.*;

class GFG{
  static int knapSack(int W, int wt[], int val[], int n)
  {
    // making and initializing dp array
    int []dp = new int[W + 1];


    for (int i = 1; i < n + 1; i++) {
      for (int w = W; w >= 0; w--) {

        if (wt[i - 1] <= w)

          // finding the maximum value
          dp[w] = Math.max(dp[w],
                           dp[w - wt[i - 1]] + val[i - 1]);

      }
    }
    return dp[W]; // returning the maximum value of knapsack
  }

  // Driver code
  public static void main(String[] args)
```

```
  {
    int val[] = { 60, 100, 120 };

    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = val.length;
    System.out.print(knapSack(W, wt, val, n));
  }
}

// This code is contributed by gauravrajput1
```

Output

```
220
```

Complexity Analysis:

Time Complexity: O(N*W). As redundant calculations of states are avoided.

Auxiliary Space: O(W) As we are using 1-D array instead of 2-D array.

Words: 1129
Characters: 6998

Post Title

Cutting a Rod | DP-13

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if the length of the rod is 8 and the values of different pieces are given as the following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 1   5   8   9  10  17  17  20
```

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 3   5   8   9  10  17  17  20
```

A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently be solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut. Let cutRod(n) be the required (best possible price) value for a rod of length n. cutRod(n) can be written as follows.

cutRod(n) = max(price[i] + cutRod(n-i-1)) for all i in {0, 1 .. n-1}

This is the approach

But Recursive not in Tabulation form.

We don't need tabulation because at end of each recursion we get our answer.

C++ Java

```
// Java recursive solution for Rod cutting problem
class GFG {

    /* Returns the best obtainable price for a rod of length
    n and price[] as prices of different pieces */
    static int cutRod(int price[], int index, int n)
    {
        // base case
        if (index == 0) {
            return n * price[0];
        }
        // At any index we have 2 options either
        // cut the rod of this length or not cut
        // it
        int notCut = cutRod(price, index - 1, n);
```

```
        int cut = Integer.MIN_VALUE;
        int rod_length = index + 1;

        if (rod_length <= n)
            cut = price[index]
                  + cutRod(price, index, n - rod_length);

        return Math.max(notCut, cut);
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
        int size = arr.length;
        System.out.println("Maximum Obtainable Value is "
                           + cutRod(arr, size - 1, size));
    }
}

// This code is contributed by Lovely Jain
```

Output

```
Maximum Obtainable Value is 22
```

2) Overlapping Subproblems

The following is a simple recursive implementation of the Rod Cutting problem.

The implementation simply follows the recursive structure mentioned above.

C++ Java

```
// A memoization solution for Rod cutting problem
import java.io.*;
import java.util.*;

/* Returns the best obtainable price for a rod of length n
and price[] as prices of different pieces */
class GFG {
  private static int cutRod(int price[], int index, int n,
                            int[][] dp)
  {

    // base case
    if (index == 0) {
      return n * price[0];
    }
```

```
    if (dp[index][n] != -1) {
      return dp[index][n];

    }

    // At any index we have 2 options either
    // cut the rod of this length or not cut
    // it
    int notCut = cutRod(price, index - 1, n, dp);
    int cut = Integer.MIN_VALUE;
    int rod_length = index + 1;

    if (rod_length <= n) {
      cut = price[index]
        + cutRod(price, index, n - rod_length,
                 dp);
    }

    return dp[index][n] = Math.max(cut, notCut);
  }

  /* Driver program to test above functions */
  public static void main(String[] args)
  {
    int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };
    int size = arr.length;
    int dp[][] = new int[size][size + 1];
    for (int i = 0; i < size; i++) {
      Arrays.fill(dp[i], -1);
    }
    System.out.println(
      "Maximum Obtainable Value is "
      + cutRod(arr, size - 1, size, dp));
  }
}

// This code is contributed by Snigdha Patil
```

Output

```
Maximum Obtainable Value is 22
```

Time Complexity: $O(n^2)$

Space Complexity: $O(n^2)+O(n)$

Considering the above implementation, the following is the recursion tree for a Rod of length 4.

```
cR() ---> cutRod()

                                cR(4)
```

Understood completely now

Approach

For each length we are taking 1 Array only and at end we got our ans.

e.g. arr[] = [1, 5, 8, 9, 10, 17, 17, 20] ← Price

Length: 1 2 3 4 5 6 7 8

Start from i=1 only.

→ For i=1 → for given capacity & Rod = 1

For j=0; j<i → (only take those rods pieces whose length is less than required capacity)

max_value = Math.max (max_value, price[j] + dp[i-j-1])

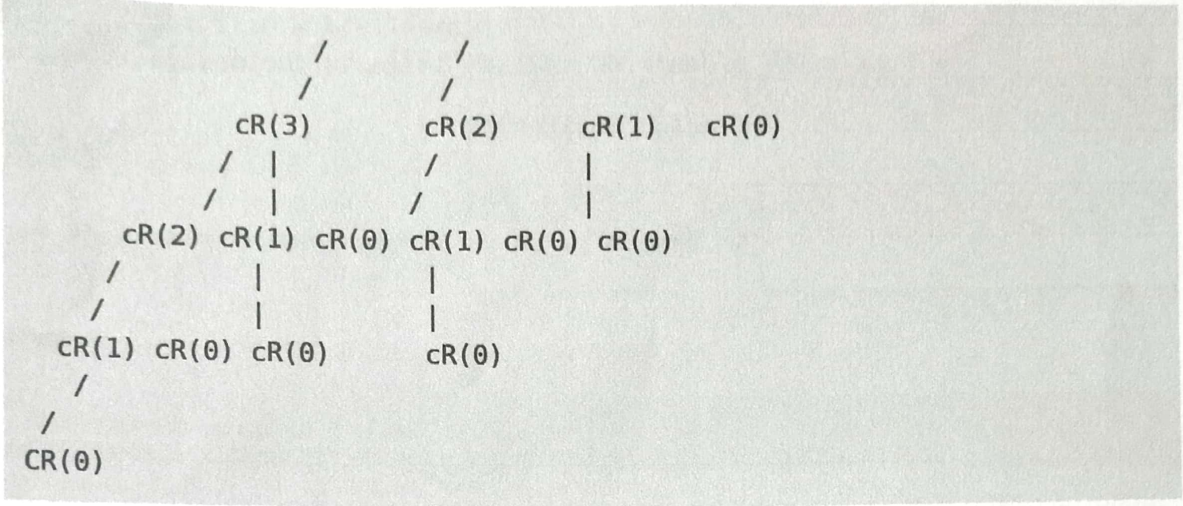
- price[j]: length of taken rod; Given j<i; j: capacity; price to be made
- dp[i-j-1]: Now from previous subproblems find the optimal ans.

OR Say

~~dp[i]~~

dp[i] = Math.max (~~max_value~~ dp[i], price[j] + dp[i-j-1])

dp[i]: initial value = 0 when creating array



In the above partial recursion tree, cR(2) is solved twice. We can see that there are many subproblems that are solved again and again. Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Rod Cutting problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array val[] in a bottom-up manner.

C++ C Java Python3 C# PHP Javascript

```
// A Dynamic Programming solution for Rod cutting problem
class RodCutting
{
    /* Returns the best obtainable price for a rod of
       length n and price[] as prices of different pieces */
    static int cutRod(int price[],int n)
    {
        int val[] = new int[n+1];
        val[0] = 0;

        // Build the table val[] in bottom up manner and return
        // the last entry from the table
        for (int i = 1; i<=n; i++)
        {
            int max_val = Integer.MIN_VALUE;
            for (int j = 0; j < i; j++)
                max_val = Math.max(max_val,
                                   price[j] + val[i-j-1]);
            val[i] = max_val;
        }

        return val[n];
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        int arr[] = new int[] {1, 5, 8, 9, 10, 17, 17, 20};
```

```
        int size = arr.length;
        System.out.println("Maximum Obtainable Value is " +
                                cutRod(arr, size));
    }
}
/* This code is contributed by Rajat Mishra */
```

Output

```
Maximum Obtainable Value is 22
```

The Time Complexity of the above implementation is O(n^2), which is much better than the worst-case time complexity of Naive Recursive implementation.

3) Using the idea of Unbounded Knapsack.

This problem is very similar to the Unbounded Knapsack Problem, where there are multiple occurrences of the same item. Here the pieces of the rod.

Now I will create an analogy between Unbounded Knapsack and the Rod Cutting Problem.

Unbounded Knapsack	Rod Cutting Problem
Size of the array(N)	Length of the Rod(Length)
Value of the item(Val)	Price of the piece of Rod(price)
Weight array(Wt[])	Length Array(length[])
Maximum weight that the Knapsack can have(W)	The length of the main rod(W)

C++ C Java Python3 C# Javascript

```
// Java program for above approach
import java.io.*;

class GFG {

    // Global Array for
    // the purpose of memoization.
    static int t[][] = new int[9][9];

    // A recursive program, using ,
    // memoization, to implement the
    // rod cutting problem(Top-Down).
    public static int un_kp(int price[], int length[],
                            int Max_len, int n)
    {
```

```
        // The maximum price will be zero,
        // when either the length of the rod
        // is zero or price is zero.
        if (n == 0 || Max_len == 0) {
            return 0;
        }

        // If the length of the rod is less
        // than the maximum length, Max_lene will
        // consider it.Now depending
        // upon the profit,
        // either Max_lene  we will take
        // it or discard it.
        if (length[n - 1] <= Max_len) {
            t[n][Max_len] = Math.max(
                price[n - 1]
                    + un_kp(price, length,
                            Max_len - length[n - 1], n),
                un_kp(price, length, Max_len, n - 1));
        }

        // If the length of the rod is
        // greater than the permitted size,
        // Max_len we will  not consider it.
        else {
            t[n][Max_len]
                = un_kp(price, length, Max_len, n - 1);
        }

        // Max_lene Max_lenill return the maximum
        // value obtained, Max_lenhich is present
        // at the nth roMax_len and Max_length column.
        return t[n][Max_len];
    }

    public static void main(String[] args)
    {

        int price[]
            = new int[] { 1, 5, 8, 9, 10, 17, 17, 20 };
        int n = price.length;
        int length[] = new int[n];
        for (int i = 0; i < n; i++) {
            length[i] = i + 1;
        }
        int Max_len = n;
        System.out.println(
            "Maximum obtained value is "
            + un_kp(price, length, n, Max_len));
    }
}
```

```
// This code is contributed by rajsanghavi9.
```

Output

```
Maximum obtained value  is 22
```

Time Complexity: $O(n^2)$

Space Complexity: O(n), since n extra space has been taken.

4) Dynamic Programming Approach Iterative Solution

We will divide the problem into smaller sub problems. Then using a 2-D matrix, we will calculate the maximum price we can achieve for any particular weight

Java

```
// Java program for above approach
import java.io.*;

class GFG {

    public static int cutRod(int prices[], int n)
    {

        int mat[][] = new int[n + 1][n + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= n; j++) {
                if (i == 0 || j == 0) {
                    mat[i][j] = 0;
                }
                else {
                    if (i == 1) {
                        mat[i][j] = j * prices[i - 1];
                    }
                    else {
                        if (i > j) {
                            mat[i][j] = mat[i - 1][j];
                        }
                        else {
                            mat[i][j] = Math.max(
                                prices[i - 1]
                                    + mat[i][j - i],
                                mat[i - 1][j]);
                        }
                    }
                }
            }
        }

        return mat[n][n];
```

```
    }
    public static void main(String[] args)

    {

        int prices[]
            = new int[] { 1, 5, 8, 9, 10, 17, 17, 20 };
        int n = prices.length;

        System.out.println("Maximum obtained value is "
                           + cutRod(prices, n));
    }
}
```

Output

```
Maximum obtained value is 22
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Words: 563
Characters: 3785

Post Title

Find the longest path in a matrix with given constraints

Given a n*n matrix where all numbers are distinct, find the maximum length path (starting from any cell) such that all cells along the path are in increasing order with a difference of 1.
We can move in 4 directions from a given cell (i, j), i.e., we can move to (i+1, j) or (i, j+1) or (i-1, j) or (i, j-1) with the condition that the adjacent cells have a difference of 1.

Example:

```
Input:  mat[][] = {{1, 2, 9}
                   {5, 3, 8}
                   {4, 6, 7}}
Output: 4
The longest path is 6-7-8-9.
```

The idea is simple, we calculate longest path beginning with every cell. Once we have computed longest for all cells, we return maximum of all longest paths. One important observation in this approach is many overlapping sub-problems. Therefore this problem can be optimally solved using Dynamic Programming.

Below is Dynamic Programming based implementation that uses a lookup table dp[][] to check if a problem is already solved or not.

C++ | Java | Python3 | Javascript | C#

```
// Java program to find the longest path in a matrix
// with given constraints

class GFG {
    public static int n = 3;

    // Function that returns length of the longest path
    // beginning with mat[i][j]
    // This function mainly uses lookup table dp[n][n]
    static int findLongestFromACell(int i, int j,
                                    int mat[][], int dp[][])
    {
        // Base case
        if (i < 0 || i >= n || j < 0 || j >= n)
            return 0;

        // If this subproblem is already solved
        if (dp[i][j] != -1)
            return dp[i][j];

        // To store the path lengths in all the four
        // directions
```

```
        // directions
        int x = Integer.MIN_VALUE, y = Integer.MIN_VALUE,
            z = Integer.MIN_VALUE, w = Integer.MIN_VALUE;

        // Since all numbers are unique and in range from 1
        // to n*n, there is atmost one possible direction
        // from any cell
        if (j < n - 1 && ((mat[i][j] + 1) == mat[i][j + 1]))
            x = dp[i][j]
                = 1
                  + findLongestFromACell(i, j + 1, mat, dp);

        if (j > 0 && (mat[i][j] + 1 == mat[i][j - 1]))
            y = dp[i][j]
                = 1
                  + findLongestFromACell(i, j - 1, mat, dp);

        if (i > 0 && (mat[i][j] + 1 == mat[i - 1][j]))
            z = dp[i][j]
                = 1
                  + findLongestFromACell(i - 1, j, mat, dp);

        if (i < n - 1 && (mat[i][j] + 1 == mat[i + 1][j]))
            w = dp[i][j]
                = 1
                  + findLongestFromACell(i + 1, j, mat, dp);

        // If none of the adjacent fours is one greater we
        // will take 1 otherwise we will pick maximum from
        // all the four directions
        return dp[i][j]
            = Math.max(
                x,
                Math.max(y, Math.max(z, Math.max(w, 1))));
    }

    // Function that returns length of the longest path
    // beginning with any cell
    static int finLongestOverAll(int mat[][])
    {
        // Initialize result
        int result = 1;

        // Create a lookup table and fill all entries in it
        // as -1
        int[][] dp = new int[n][n];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                dp[i][j] = -1;

        // Compute longest path beginning from all cells
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (dp[i][j] == -1)
```

This is top down approach used to implement recursive or backtracking approach.

```
                findLongestFromACell(i, j, mat, dp);

            // Update result if needed
            result = Math.max(result, dp[i][j]);
        }
    }

    return result;
}

// driver program
public static void main(String[] args)
{
    int mat[][]
        = { { 1, 2, 9 }, { 5, 3, 8 }, { 4, 6, 7 } };
    System.out.println("Length of the longest path is "
                       + finLongestOverAll(mat));
}
}

// Contributed by Pramod Kumar
```

Output

```
Length of the longest path is 4
```

Time complexity of the above solution is $O(n^2)$. It may seem more at first look. If we take a closer look, we can notice that all values of dp[i][j] are computed only once.
Auxiliary Space: O(N x N), since N x N extra space has been taken.

This article is contributed by **Aarti_Rathi** and Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Words: 246
Characters: 1400