Dynamic Programming

[**Learn more about Dynamic Programming in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+dp_lp+header_link_click&utm_campaign=dsa+course+tracker)

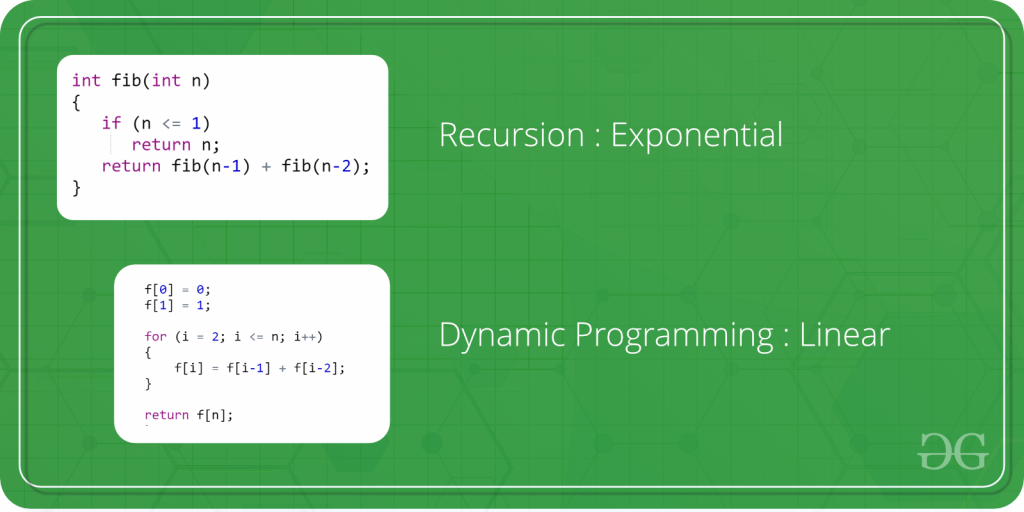
[**Practice Problems on Dynamic Programming**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Dynamic%20Programming&page=1&category%5B%5D=Dynamic%20Programming&utm_source=geeksforgeeks&utm_medium=articles+dp_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Dynamic Programming**](https://www.geeksforgeeks.org/category/algorithm/dynamic-programming/)

[**What is Dynamic Programming?**](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/)

Dynamic Programming is mainly an optimization over plain [recursion](https://www.geeksforgeeks.org/recursion/). Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

For example, if we write simple recursive solution for [Fibonacci Numbers](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/), we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



**Topics:**

* [Basic Concepts](https://www.geeksforgeeks.org/dynamic-programming/#basics)
* [Advanced Concepts](https://www.geeksforgeeks.org/dynamic-programming/#advanced)
* [Standard Dynamic Programming problems](https://www.geeksforgeeks.org/dynamic-programming/#standard)
* [Quick Links](https://www.geeksforgeeks.org/dynamic-programming/#quick)

**Basic Concepts:**

1. [What is memoization? A Complete tutorial](https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/)
2. [Introduction to Dynamic Programming – Data Structures and Algorithm Tutorials](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/)
3. [Tabulation vs Memoizatation](https://www.geeksforgeeks.org/tabulation-vs-memoizatation/)
4. [Optimal Substructure Property](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)
5. [Overlapping Subproblems Property](https://www.geeksforgeeks.org/dynamic-programming-set-1/)
6. [How to solve a Dynamic Programming Problem ?](https://www.geeksforgeeks.org/solve-dynamic-programming-problem/)

**Advanced Concepts:**

1. [Bitmasking and Dynamic Programming | Set 1](https://www.geeksforgeeks.org/bitmasking-and-dynamic-programming-set-1-count-ways-to-assign-unique-cap-to-every-person/)
2. [Bitmasking and Dynamic Programming | Set-2 (TSP)](https://www.geeksforgeeks.org/bitmasking-dynamic-programming-set-2-tsp/)
3. [Digit DP | Introduction](https://www.geeksforgeeks.org/digit-dp-introduction/)
4. [Sum over Subsets | Dynamic Programming](https://www.geeksforgeeks.org/sum-subsets-dynamic-programming/)

**Standard problems on Dynamic Programming:**

* **Easy:**
  1. [Fibonacci numbers](https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/)
  2. [nth Catalan Number](https://www.geeksforgeeks.org/program-nth-catalan-number/)
  3. [Bell Numbers (Number of ways to Partition a Set)](https://www.geeksforgeeks.org/bell-numbers-number-of-ways-to-partition-a-set/)
  4. [Binomial Coefficient](https://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/)
  5. [Coin change problem](https://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/)
  6. [Subset Sum Problem](https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/)
  7. [Compute nCr % p](https://www.geeksforgeeks.org/compute-ncr-p-set-1-introduction-and-dynamic-programming-solution/)
  8. [Cutting a Rod](https://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/)
  9. [Painting Fence Algorithm](https://www.geeksforgeeks.org/painting-fence-algorithm/)
  10. [Longest Common Subsequence](https://www.geeksforgeeks.org/longest-common-subsequence/)
  11. [Longest Increasing Subsequence](https://www.geeksforgeeks.org/longest-increasing-subsequence/)
  12. [Longest subsequence such that difference between adjacents is one](https://www.geeksforgeeks.org/longest-subsequence-such-that-difference-between-adjacents-is-one/)
  13. [Maximum size square sub-matrix with all 1s](https://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/)
  14. [Min Cost Path](https://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/)
  15. [Minimum number of jumps to reach end](https://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/)
  16. [Longest Common Substring (Space optimized DP solution)](https://www.geeksforgeeks.org/longest-common-substring-space-optimized-dp-solution/)
  17. [Count ways to reach the nth stair using step 1, 2 or 3](https://www.geeksforgeeks.org/count-ways-reach-nth-stair-using-step-1-2-3/)
  18. [Count all possible paths from top left to bottom right of a mXn matrix](https://www.geeksforgeeks.org/count-possible-paths-top-left-bottom-right-nxm-matrix/)
  19. [Unique paths in a Grid with Obstacles](https://www.geeksforgeeks.org/unique-paths-in-a-grid-with-obstacles/)
* **Medium:**
  1. [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
  2. [Bellman–Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
  3. [0-1 Knapsack Problem](https://www.geeksforgeeks.org/knapsack-problem/)
  4. [Printing Items in 0/1 Knapsack](https://www.geeksforgeeks.org/printing-items-01-knapsack/)
  5. [Unbounded Knapsack (Repetition of items allowed)](https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed/)
  6. [Egg Dropping Puzzle](https://www.geeksforgeeks.org/dynamic-programming-set-11-egg-dropping-puzzle/)
  7. [Word Break Problem](https://www.geeksforgeeks.org/dynamic-programming-set-32-word-break-problem/)
  8. [Vertex Cover Problem](https://www.geeksforgeeks.org/vertex-cover-problem-set-2-dynamic-programming-solution-tree/)
  9. [Tile Stacking Problem](https://www.geeksforgeeks.org/tile-stacking-problem/)
  10. [Box-Stacking Problem](https://www.geeksforgeeks.org/dynamic-programming-set-21-box-stacking-problem/)
  11. [Partition Problem](https://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/)
  12. [Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/)
  13. [Longest Palindromic Subsequence](https://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/)
  14. [Longest Common Increasing Subsequence (LCS + LIS)](https://www.geeksforgeeks.org/longest-common-increasing-subsequence-lcs-lis/)
  15. [Find all distinct subset (or subsequence) sums of an array](https://www.geeksforgeeks.org/find-distinct-subset-subsequence-sums-array/)
  16. [Weighted job scheduling](https://www.geeksforgeeks.org/weighted-job-scheduling/)
  17. [Count Derangements (Permutation such that no element appears in its original position)](https://www.geeksforgeeks.org/count-derangements-permutation-such-that-no-element-appears-in-its-original-position/)
  18. [Minimum insertions to form a palindrome](https://www.geeksforgeeks.org/dynamic-programming-set-28-minimum-insertions-to-form-a-palindrome/)
  19. [Wildcard Pattern Matching](https://www.geeksforgeeks.org/wildcard-pattern-matching/)
  20. [Ways to arrange Balls such that adjacent balls are of different types](https://www.geeksforgeeks.org/ways-to-arrange-balls-such-that-adjacent-balls-are-of-different-types/)
* **Hard:**
  1. [Palindrome Partitioning](https://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/)
  2. [Word Wrap Problem](https://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/)
  3. [The painter’s partition problem](https://www.geeksforgeeks.org/painters-partition-problem/)
  4. [Program for Bridge and Torch problem](https://www.geeksforgeeks.org/program-bridge-torch-problem/)
  5. [Matrix Chain Multiplication](https://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/)
  6. [Printing brackets in Matrix Chain Multiplication Problem](https://www.geeksforgeeks.org/printing-brackets-matrix-chain-multiplication-problem/)
  7. [Maximum sum rectangle in a 2D matrix](https://www.geeksforgeeks.org/dynamic-programming-set-27-max-sum-rectangle-in-a-2d-matrix/)
  8. [Maximum profit by buying and selling a share at most k times](https://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-k-times/)
  9. [Minimum cost to sort strings using reversal operations of different costs](https://www.geeksforgeeks.org/minimum-cost-sort-strings-using-reversal-operations-different-costs/)
  10. [Count of AP (Arithmetic Progression) Subsequences in an array](https://www.geeksforgeeks.org/count-arithmetic-progression-subsequences-array/)
  11. [Introduction to Dynamic Programming on Trees](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-on-trees/)
  12. [Maximum height of Tree when any Node can be considered as Root](https://www.geeksforgeeks.org/maximum-height-of-tree-when-any-node-can-be-considered-as-root/)
  13. [Longest repeating and non-overlapping substring](https://www.geeksforgeeks.org/longest-repeating-and-non-overlapping-substring/)

*-------------------------------------------------------------------------------------------------------------------------------------------------------------*

**Easy Questions:**

**1.Program for Fibonacci numbers**

The Fibonacci numbers are the numbers in the following integer sequence.

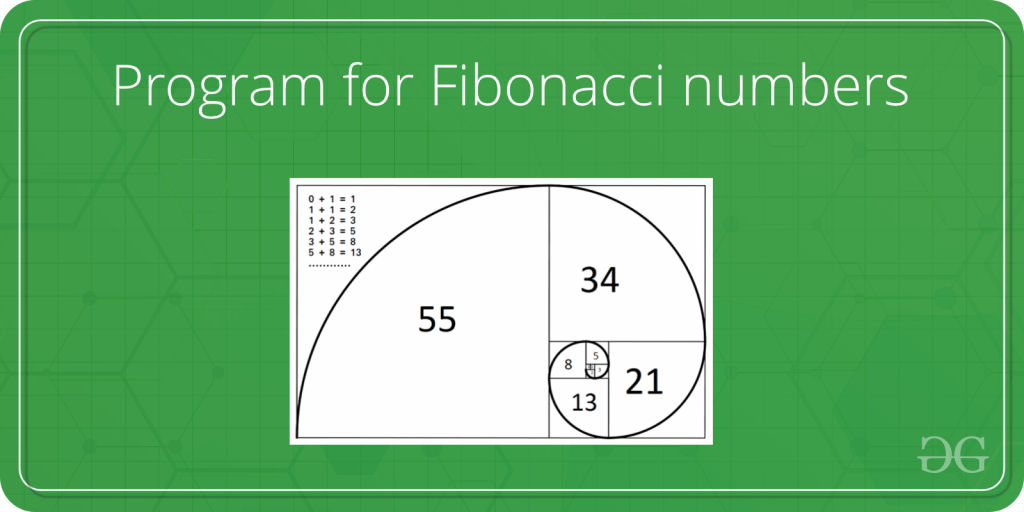
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

Fn = Fn-1 + Fn-2

with seed values

F0 = 0 and F1 = 1.



Given a number n, print n-th Fibonacci Number.

**Examples:**

Input : n = 2  
Output : 1

Input : n = 9  
Output : 34

Nth Fibonacci Number

Write a function *int fib(int n)* that returns Fn. For example, if *n* = 0, then *fib()* should return 0. If n = 1, then it should return 1. For n > 1, it should return Fn-1 + Fn-2

For n = 9  
Output:34

The following are different methods to get the nth Fibonacci number.

**Method 1 (Use recursion)**

A simple method that is a direct recursive implementation mathematical recurrence relation is given above.

# Fibonacci series using recursion

**def** fibonacci(n):

**if** n <**=** 1:

**return** n

**return** fibonacci(n**-**1) **+** fibonacci(n**-**2)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    n **=** 9

    print(fibonacci(n))

 # This code is contributed by Manan Tyagi.

**Output**

34

**Time Complexity: Exponential,** as every function calls two other functions.

If the original recursion tree were to be implemented then this would have been the tree but now for n times the recursion function is called

Original tree for recursion

fib(5)   
 / \  
 fib(4) fib(3)   
 / \ / \   
 fib(3) fib(2) fib(2) fib(1)  
 / \ / \ / \  
 fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)  
 / \  
fib(1) fib(0)

Optimized tree for recursion for code above

    fib(5)

    fib(4)

    fib(3)

    fib(2)

    fib(1)

*Extra Space:* O(n) if we consider the function call stack size, otherwise O(1).

**Method 2: (Use Dynamic Programming)**

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

# Fibonacci Series using Dynamic Programming

**def** fibonacci(n):

    # Taking 1st two fibonacci numbers as 0 and 1

    f **=** [0, 1]

**for** i **in** range(2, n**+**1):

        f.append(f[i**-**1] **+** f[i**-**2])

**return** f[n]

print(fibonacci(9))

**Output**

34

**Time complexity**: O(n) for given n

**Auxiliary space**: O(n)

**Method 3: (Space Optimized Method 2)**

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

# Function for nth fibonacci number - Space Optimisation

# Taking 1st two fibonacci numbers as 0 and 1

**def** fibonacci(n):

    a **=** 0

    b **=** 1

**if** n < 0:

        print("Incorrect input")

**elif** n **==** 0:

**return** a

**elif** n **==** 1:

**return** b

**else**:

**for** i **in** range(2,n**+**1):

            c **=** a **+** b

            a **=** b

            b **=** c

**return** b

# Driver Program

print(fibonacci(9))

#This code is contributed by Saket Modi

**Output**

34

**Time Complexity:**O(n)

**Extra Space:**O(1)

**Method 4: Using power of the matrix {{1, 1}, {1, 0}}**

This is another O(n) that relies on the fact that if we n times multiply the matrix M = {{1,1},{1,0}} to itself (in other words calculate power(M, n)), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

# Helper function that multiplies

# 2 matrices F and M of size 2\*2,

# and puts the multiplication

# result back to F[][]

# Helper function that calculates

# F[][] raise to the power n and

# puts the result in F[][]

# Note that this function is

# designed only for fib() and

# won't work as general

# power function

**def** fib(n):

    F **=** [[1, 1],

         [1, 0]]

**if** (n **==** 0):

**return** 0

    power(F, n **-** 1)

**return** F[0][0]

**def** multiply(F, M):

    x **=** (F[0][0] **\*** M[0][0] **+**

         F[0][1] **\*** M[1][0])

    y **=** (F[0][0] **\*** M[0][1] **+**

         F[0][1] **\*** M[1][1])

    z **=** (F[1][0] **\*** M[0][0] **+**

         F[1][1] **\*** M[1][0])

    w **=** (F[1][0] **\*** M[0][1] **+**

         F[1][1] **\*** M[1][1])

    F[0][0] **=** x

    F[0][1] **=** y

    F[1][0] **=** z

    F[1][1] **=** w

**def** power(F, n):

    M **=** [[1, 1],

         [1, 0]]

    # n - 1 times multiply the

    # matrix to {{1,0},{0,1}}

**for** i **in** range(2, n **+** 1):

        multiply(F, M)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    n **=** 9

    print(fib(n))

# This code is contributed

# by ChitraNayal

**Output**

34

**Time Complexity:** O(n)

**Auxiliary Space:**O(1)

**Method 5: (Optimized Method 4)**

Method 4 can be optimized to work in O(Logn) time complexity. We can do recursive multiplication to get power(M, n) in the previous method (Similar to the optimization done in [this](https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/)post)

# Fibonacci Series using

# Optimized Method

# function that returns nth

# Fibonacci number

**def** fib(n):

    F **=** [[1, 1],

         [1, 0]]

**if** (n **==** 0):

**return** 0

    power(F, n **-** 1)

**return** F[0][0]

**def** multiply(F, M):

    x **=** (F[0][0] **\*** M[0][0] **+**

         F[0][1] **\*** M[1][0])

    y **=** (F[0][0] **\*** M[0][1] **+**

         F[0][1] **\*** M[1][1])

    z **=** (F[1][0] **\*** M[0][0] **+**

         F[1][1] **\*** M[1][0])

    w **=** (F[1][0] **\*** M[0][1] **+**

         F[1][1] **\*** M[1][1])

    F[0][0] **=** x

    F[0][1] **=** y

    F[1][0] **=** z

    F[1][1] **=** w

# Optimized version of

# power() in method 4

**def** power(F, n):

**if**( n **==** 0 **or** n **==** 1):

**return**;

    M **=** [[1, 1],

         [1, 0]];

    power(F, n **//** 2)

    multiply(F, F)

**if** (n **%** 2 !**=** 0):

        multiply(F, M)

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    n **=** 9

    print(fib(n))

# This code is contributed

# by ChitraNayal

**Output**

34

**Time Complexity:O(Logn)**

**Auxiliary Space:** O(Logn) if we consider the function call stack size, otherwise O(1).

**Method 6: (O(Log n) Time)**

Below is one more interesting recurrence formula that can be used to find n’th Fibonacci Number in O(Log n) time.

If n is even then k = n/2:  
F(n) = [2\*F(k-1) + F(k)]\*F(k)

If n is odd then k = (n + 1)/2  
F(n) = F(k)\*F(k) + F(k-1)\*F(k-1)

**How does this formula work?**

The formula can be derived from the above matrix equation.

Taking determinant on both sides, we get

(-1)n = Fn+1Fn-1 - Fn2   
   
Moreover, since AnAm = An+m for any square matrix A,   
the following identities can be derived (they are obtained   
from two different coefficients of the matrix product)

FmFn + Fm-1Fn-1 = Fm+n-1 ---------------------------(1)

By putting n = n+1 in equation(1),  
FmFn+1 + Fm-1Fn = Fm+n --------------------------(2)

Putting m = n in equation(1).  
F2n-1 = Fn2 + Fn-12  
Putting m = n in equation(2)

F2n = (Fn-1 + Fn+1)Fn = (2Fn-1 + Fn)Fn (Source: [Wiki](https://en.wikipedia.org/wiki/Fibonacci_number#Matrix_form)) --------  
( By putting Fn+1 = Fn + Fn-1 )  
To get the formula to be proved, we simply need to do the following   
If n is even, we can put k = n/2   
If n is odd, we can put k = (n+1)/2

Below is the implementation of the above idea.

# Python3 Program to find n'th fibonacci Number in

# with O(Log n) arithmetic operations

MAX **=** 1000

# Create an array for memoization

f **=** [0] **\*** MAX

# Returns n'th fibonacci number using table f[]

**def** fib(n) :

    # Base cases

**if** (n **==** 0) :

**return** 0

**if** (n **==** 1 **or** n **==** 2) :

        f[n] **=** 1

**return** (f[n])

    # If fib(n) is already computed

**if** (f[n]) :

**return** f[n]

**if**( n & 1) :

        k **=** (n **+** 1) **//** 2

**else** :

        k **=** n **//** 2

    # Applying above formula [Note value n&1 is 1

    # if n is odd, else 0.

**if**((n & 1) ) :

        f[n] **=** (fib(k) **\*** fib(k) **+** fib(k**-**1) **\*** fib(k**-**1))

**else** :

        f[n] **=** (2**\***fib(k**-**1) **+** fib(k))**\***fib(k)

**return** f[n]

# Driver code

n **=** 9

print(fib(n))

# This code is contributed by Nikita Tiwari.

**Output**

34

**Time Complexity:** O(Log n), as we divide the problem in half in every recursive call.

**Auxiliary Space:** O(n)

**Method 7: (Another approach(Using Binet’s formula))**

In this method, we directly implement the formula for the nth term in the Fibonacci series.

Fn = {[(√5 + 1)/2] ^ n} / √5

***Note:****Above Formula gives correct result only upto for n<71. Because as we move forward from n>=71 , rounding error becomes significantly large . Although , using floor function instead of round function will give correct result for n=71 . But after from n=72 , it also fails.*

***Example: For N=72 , Correct result is 498454011879264 but above formula gives 498454011879265.***

Reference: <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibFormula.html>

# Python3 program to find n'th

# fibonacci Number

**import** math

**def** fibo(n):

    phi **=** (1 **+** math.sqrt(5)) **/** 2

**return** round(pow(phi, n) **/** math.sqrt(5))

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    n **=** 9

    print(fibo(n))

# This code is contributed by prasun\_parate

**Output**

34

**Time Complexity*:***O(logn), this is because calculating phi^n takes logn time

**Auxiliary Space*:***O(1)

**Method 8: DP using memoization(Top down approach)**

We can avoid the repeated work done in method 1 by storing the Fibonacci numbers calculated so far. We just need to store all the values in an array.

# Initialize array of dp

dp **=** [**-**1 **for** i **in** range(10)]

**def** fib(n):

**if** (n <**=** 1):

**return** n;

**global** dp;

    # Temporary variables to store

    # values of fib(n-1) & fib(n-2)

    first **=** 0;

    second **=** 0;

**if** (dp[n **-** 1] !**= -**1):

        first **=** dp[n **-** 1];

**else**:

        first **=** fib(n **-** 1);

**if** (dp[n **-** 2] !**= -**1):

        second **=** dp[n **-** 2];

**else**:

        second **=** fib(n **-** 2);

    dp[n] **=** first **+** second;

    # Memoization

**return** dp[n] ;

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    n **=** 9;

    print(fib(n));

# This code contributed by Rajput-Ji

**Output**

34

**Time Complexity:** O(n)

**Auxiliary Space:** O(n)

**Related Articles:**

[Large Fibonacci Numbers in Java](https://www.geeksforgeeks.org/large-fibonacci-numbers-java/)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

**References:**

<http://en.wikipedia.org/wiki/Fibonacci_number>

<http://www.ics.uci.edu/~eppstein/161/960109.html>

**Method 9 : (Kartik’s K sequence) here K=3**

1)  **0**,1,1,**2**,3,5,**8**,13,21,**34**,55,89,**144**,….. (Parallel 0 highlighted with Bold)

2)  0,**1**,1,2,**3**,5,8,**13**,21,34,**55**,89,144,….. (Parallel 1 highlighted with Bold)

3)  0,1,**1**,2,3,**5**,8,13,**21**,34,55,**89**,144,….. (Parallel 2 highlighted with Bold)

**if you observed the bold Numbers**

**consider Parallel 0  bold Number form 1)**

**0,2,8,34,144,…**

*2 \* 4  + 0 = 8  (7th)*

*8 \* 4 + 2 = 34  (10th)*

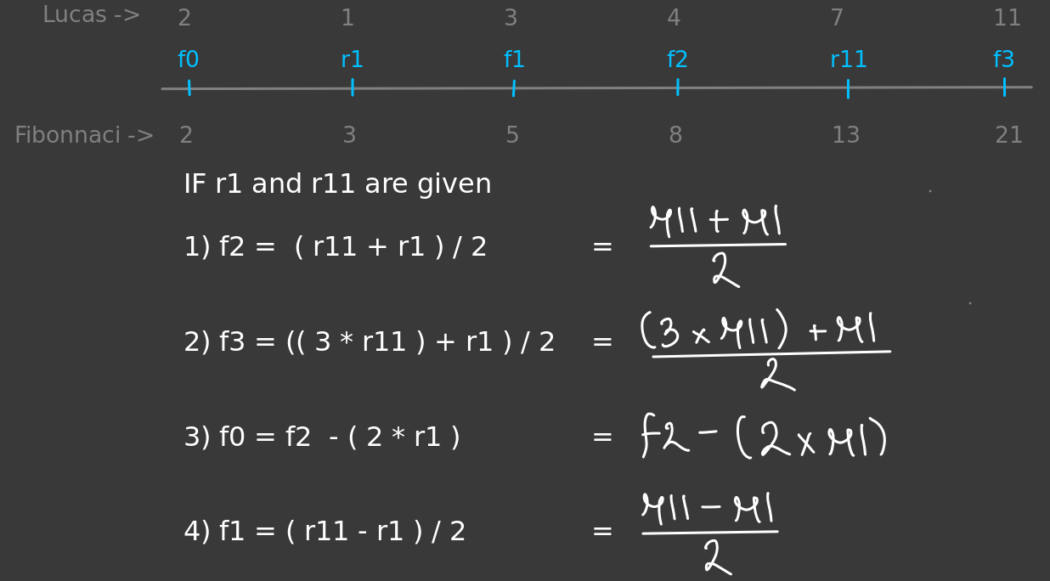
*34 \* 4 + 8 = 144  (13th)*

**N+1th \* 4 + Nth = N+2th which can be applied to all three Parallel**

**and shifting rules**

**using for F1 and F2 it can be replicated to Lucas sequence as well**

**in the below  image “Parallel 1” as r1 = 3 and r11=13**



***Shifting***

**AND HERE I USED  1,1,5,21,… Parallel 2**

**def** nth\_fibonnaci(n):

**if** n > 0:

        n1, n2 **=** 1, 1

**if** n > 3:

**for** \_ **in** range((n**//**3)):

                n1, n2 **=** n2, (n2 << 2)**+**n1  # << 2   is multiply by 4

**if** n **%** 3 **==** 0:

**return** n1

**elif** n **%** 3 **==** 1:

**return** (n2**-**n1) >> 1  # >> 1   is divide by 2  'F1'

**elif** n **%** 3 **==** 2:

**return** (n2**+**n1) >> 1  # >> 1   is divide by 2  'F2'

**else**:

**return -**1

**for** i **in** range(1, 9):

**print**(f"{nth\_fibonnaci(i)} is {i}", end**=**"th  | ")

print("")

**for** i **in** range(9, 30, 3):

    print(f"{nth\_fibonnaci(i)} is {i}", end**=**"th  | ")

**Output**

0 is 1th | 1 is 2th | 1 is 3th | 2 is 4th | 3 is 5th | 5 is 6th | 8 is 7th | 13 is 8th |   
21 is 9th | 89 is 12th | 377 is 15th | 1597 is 18th | 6765 is 21th | 28657 is 24th | 121393 is 27th |

**Time Complexity: in between O(log n) and O(n) or  (n/3)**

**Auxiliary Space: O(1) (constant)**

**Related Articles:**

<https://medium.com/@kartikmoyade0901/something-new-for-maths-and-it-researchers-or-professional-1df60058485d>

**3.Program for nth Catalan Number**

**Catalan numbers** are defined as a mathematical sequence that consists of positive integers, which can be used to find the number of possibilities of various combinations.

The **nth**term in the sequence denoted**Cn**, is found in the following formula:

*The first few Catalan numbers for n = 0, 1, 2, 3, … are : 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …*

**Catalan numbers occur in many interesting counting problems like the following.**

1. Count the number of expressions containing n pairs of parentheses that are correctly matched. For n = 3, possible expressions are **((())), ()(()), ()()(), (())(), (()())**.
2. Count the number of possible Binary Search Trees with**n** keys (See [this](https://www.geeksforgeeks.org/g-fact-18/))
3. Count the number of full binary trees (A rooted binary tree is full if every vertex has either two children or no children) with **n+1**leaves.
4. Given a number **n**, return the number of ways you can draw n chords in a circle with **2 x n** points such that no **2** chords intersect.

**See**[**this**](https://www.geeksforgeeks.org/applications-of-catalan-numbers/)**for more applications.**

**Examples:**

***Input:****n = 6*

***Output:****132*

***Input:****n = 8*

***Output:****1430*

[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/nth-catalan-number0817/1)

**4.Recursive Solution for Catalan number:**

*Catalan numbers satisfy the following recursive formula:*

Follow the steps below to implement the above recursive formula

* Base condition for the recursive approach, when **n <= 1**, return **1**
* Iterate from i = 0 to i < n
* Make a recursive call**catalan(i)**and**catalan(n – i – 1)** and keep adding the product of both into **res**.
* Return the **res**.

Following is the implementation of the above recursive formula.

# A recursive function to

# find nth catalan number

**def** catalan(n):

    # Base Case

**if** n <**=** 1:

**return** 1

    # Catalan(n) is the sum

    # of catalan(i)\*catalan(n-i-1)

    res **=** 0

**for** i **in** range(n):

        res **+=** catalan(i) **\*** catalan(n**-**i**-**1)

**return** res

# Driver Code

**for** i **in** range(10):

    print(catalan(i), end**=**" ")

# This code is contributed by

# Nikhil Kumar Singh (nickzuck\_007)

**Output**

1 1 2 5 14 42 132 429 1430 4862

**Time Complexity:** The above implementation is equivalent to nth Catalan number.

The value of **nth**Catalan number is exponential which makes the time complexity exponential.

**Auxiliary Space:**O(n)

**5.Dynamic Programming Solution for Catalan number:**

*We can observe that the above recursive implementation does a lot of repeated work. Since there are overlapping subproblems, we can use dynamic programming for this.*

**Below is the implementation of the above idea:**

* Create an array **catalan[]**for storing **ith**Catalan number.
* Initialize, **catalan[0] and catalan[1] = 1**
* Loop through **i = 2** to the given Catalan number **n**.
* Loop through**j = 0**to **j < i** and Keep adding value of **catalan[j] \* catalan[i – j – 1]** into **catalan[i]**.
* Finally, return **catalan[n]**

Follow the steps below to implement the above approach:

# A dynamic programming based function to find nth

# Catalan number

**def** catalan(n):

**if** (n **==** 0 **or** n **==** 1):

**return** 1

    # Table to store results of subproblems

    catalan **=** [0]**\***(n**+**1)

    # Initialize first two values in table

    catalan[0] **=** 1

    catalan[1] **=** 1

    # Fill entries in catalan[]

    # using recursive formula

**for** i **in** range(2, n **+** 1):

**for** j **in** range(i):

            catalan[i] **+=** catalan[j] **\*** catalan[i**-**j**-**1]

    # Return last entry

**return** catalan[n]

# Driver code

**for** i **in** range(10):

**print**(catalan(i), end**=**" ")

# This code is contributed by Ediga\_manisha

**Output**

1 1 2 5 14 42 132 429 1430 4862

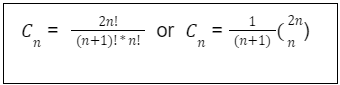
**Time Complexity:** O(n2)

**Auxiliary Space:** O(n)

**6.Binomial Coefficient  Solution for Catalan number:**

*We can also use the below formula to find****nth****Catalan number in****O(n)****time.*

Here is the proof for the Expression:–



***This is the expression for which we are going to see the proof***

In the pascal triangle,

                           1

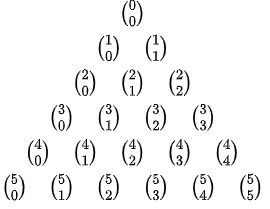
                      1         1

                 1        2        1

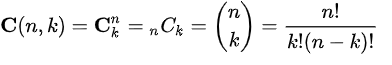
             1       3         3       1

        1      4         6       4         1

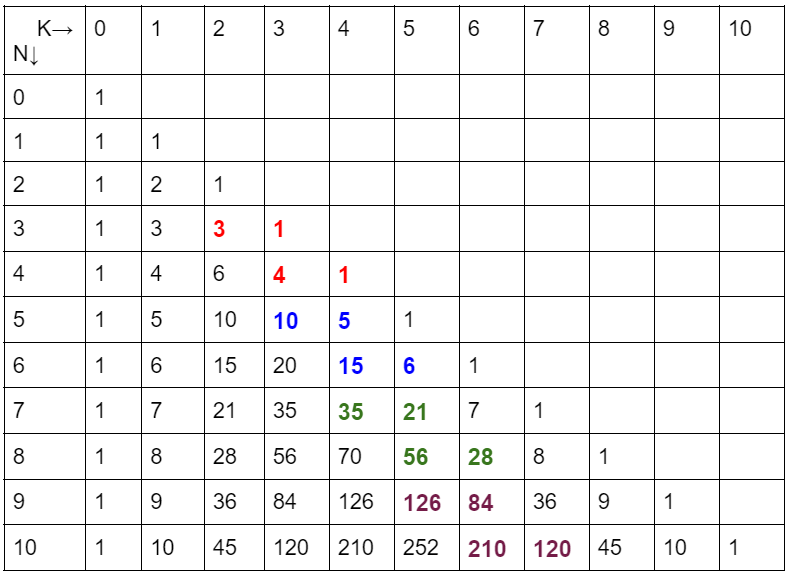
   1     5         10     10       5         1



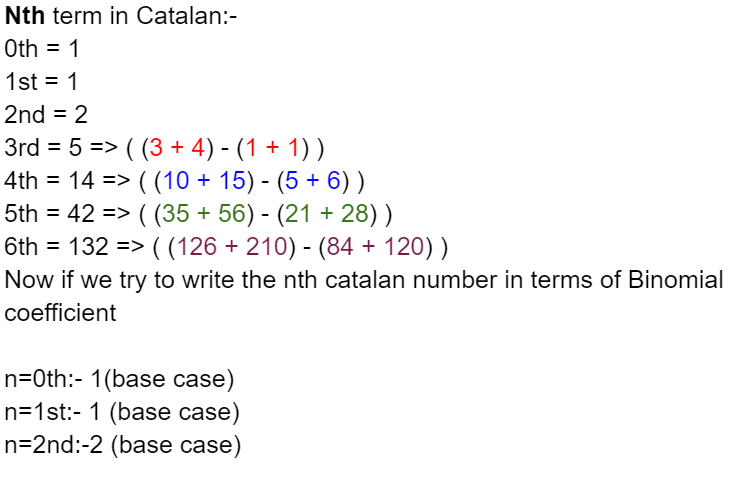
***Pascal’s triangle as binomial coefficients***



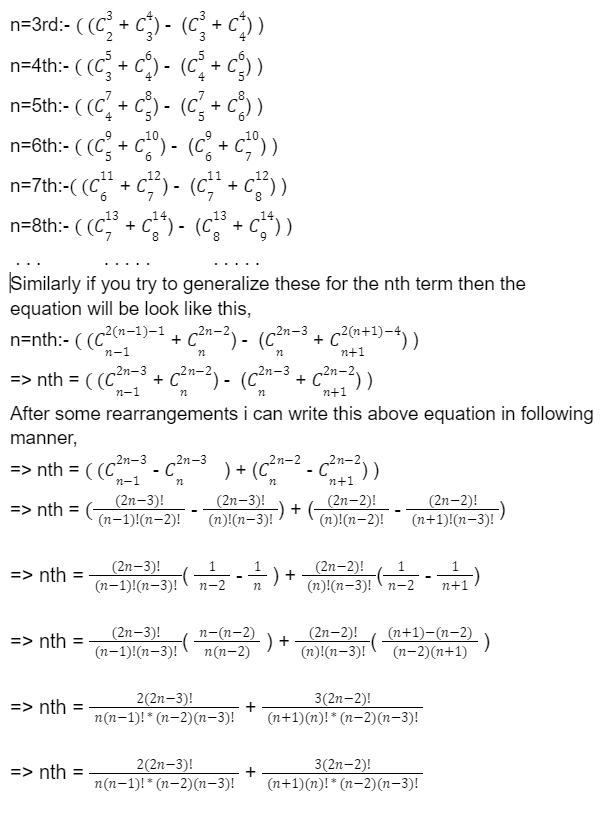
***the formula for a cell of Pascal’s triangle***



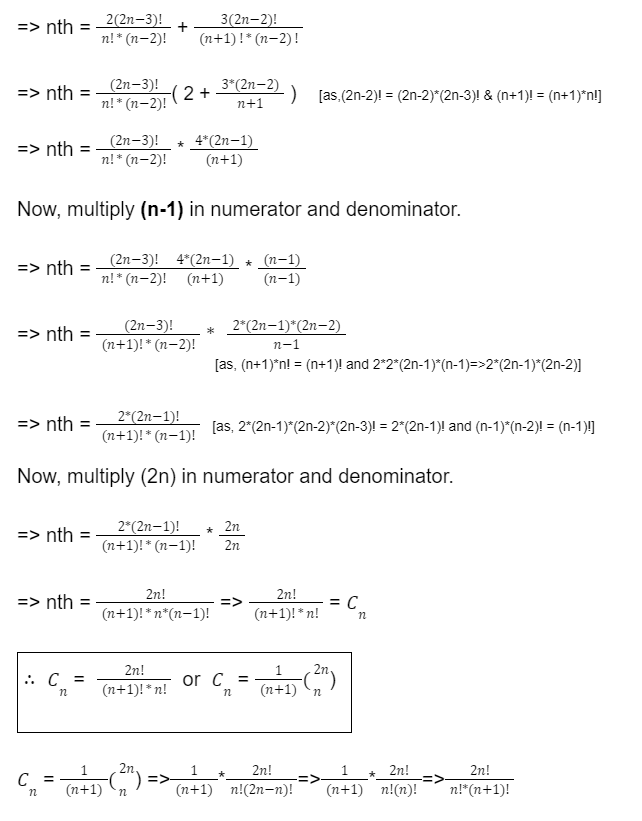
***Pascal’s triangle***



*.*



*.*



*.*

Below are the steps for calculating**nCr**.

* Create a variable to store the answer and change **r** to **n – r** if **r** is greater than **n – r** because we know that **C(n, r) = C(n, n-r)**if r > n – r
* Run a loop from **0** to **r-1**
* In every iteration update ans as**(ans\*(n-i))/(i+1)**, where i is the loop counter.
* So the answer will be equal to**((n/1)\*((n-1)/2)\*…\*((n-r+1)/r)**, which is equal to **nCr**.

Below are steps to calculate Catalan numbers using the formula: 2nCn/(n+1)

* Calculate **2nCn** using the similar steps that we use to calculate**nCr**
* Return the value 2nCn/ (n + 1)\

Below is the implementation of the above approach:

# Python program for nth Catalan Number

# Returns value of Binomial Coefficient C(n, k)

**def** binomialCoefficient(n, k):

    # since C(n, k) = C(n, n - k)

**if** (k > n **-** k):

        k **=** n **-** k

    # initialize result

    res **=** 1

    # Calculate value of [n \* (n-1) \*---\* (n-k + 1)]

    # / [k \* (k-1) \*----\* 1]

**for** i **in** range(k):

        res **=** res **\*** (n **-** i)

        res **=** res **/** (i **+** 1)

**return** res

# A Binomial coefficient based function to

# find nth catalan number in O(n) time

**def** catalan(n):

    c **=** binomialCoefficient(2**\***n, n)

**return** c**/**(n **+** 1)

# Driver Code

**for** i **in** range(10):

    print(catalan(i), end**=**" ")

# This code is contributed by Aditi Sharma

**Output**

1 1 2 5 14 42 132 429 1430 4862

**Time Complexity:** O(n).

**Auxiliary Space:** O(1)

We can also use the below formulas to find **nth**Catalan number in O(n) time.

**7.Catalan number using the multi-Precision library:**

In this method, we have used a boost **multi-precision library**, and the motive behind its use is just only to have precision meanwhile finding the large Catalan number and a generalized technique using for loop to calculate Catalan numbers.

**Pseudocode:**

a) initially set cat\_=1 and print it  
b) run a for loop i=1 to i<=n  
 cat\_ \*= (4\*i-2)  
 cat\_ /= (i+1)  
 print cat\_  
c) end loop and exit

Below is the implementation using the multi-precision library:

# Function to print the number

**def** catalan(n):

    cat\_ **=** 1

    # For the first number

**print**(cat\_, " ", end**=**'')  # C(0)

    # Iterate till N

**for** i **in** range(1, n):

        # Calculate the number

        # and print it

        cat\_ **\*=** (4 **\*** i **-** 2)

        cat\_ **//=** (i **+** 1)

        print(cat\_, " ", end**=**'')

# Driver code

n **=** 5

# Function call

catalan(n)

# This code is contributed by rohan07

**Output**

1 1 2 5 14

**Time Complexity:**O(n)

**Auxiliary Space:**O(1), since no extra space has been taken.

**8.Catalan number using BigInteger in java:**

Finding values of Catalan numbers for **N>80** is not possible even by using **long**in java, so we use **BigInteger**.

Follow the steps below for the implementation:

* Create a BigInteger variable **b**and initialize it to **1**.
* Calculate **n!**and store it into **b**.
* Calculate **n! \* n!** and store into **b**.
* Create another BigInteger variable **d**and initialize it to **1**.
* Calculate **2n!**and store into **d**.
* Calculate **(2n)! / (n! \* n!)** into **ans**
* Calculate **ans / (n + 1)**and return **ans**.

Below is the implementation of the above approach:

**def** findCatalan(n):

    b **=** 1

    # calculating n!

**for** i **in** range(1, n **+** 1, 1):

        b **=** b **\*** i

    # calculating n! \* n!

    b **=** b **\*** b

    d **=** 1

    # calculating (2n)!

**for** i **in** range(1, 2 **\*** n **+** 1, 1):

        d **=** d **\*** i

    # calculating (2n)! / (n! \* n!)

    ans **=** d **/** b

    # calculating (2n)! / ((n! \* n!) \* (n+1))

    ans **=** ans **/** (n **+** 1)

**return** ans

# Driver Code

n **=** 50

print(int(findCatalan(n)))

# This code is contributed by ajaymakavana.

**Output**

42

**Time Complexity:**O(n)

**Auxiliary Space:**O(1), since no extra space has been taken.

**9.Bell Numbers (Number of ways to Partition a Set)**

Given a set of n elements, find number of ways of partitioning it.

**Examples:**

Input: n = 2  
Output: Number of ways = 2  
Explanation: Let the set be {1, 2}  
 { {1}, {2} }   
 { {1, 2} }

Input: n = 3  
Output: Number of ways = 5  
Explanation: Let the set be {1, 2, 3}  
 { {1}, {2}, {3} }  
 { {1}, {2, 3} }  
 { {2}, {1, 3} }  
 { {3}, {1, 2} }  
 { {1, 2, 3} }.

[Recommended practice](https://practice.geeksforgeeks.org/problems/bell-numbers2108/1/?page=2&difficulty%5b%5d=0&status%5b%5d=unsolved&category%5b%5d=Dynamic%20Programming&sortBy=submissions)

The solution to above questions is [Bell Number](https://en.wikipedia.org/wiki/Bell_number).

**What is a Bell Number?**

Let **S(n, k)** be total number of partitions of n elements into k sets. The value of n’th Bell Number is sum of S(n, k) for k = 1 to n.

Value of S(n, k) can be defined recursively as, S(n+1, k) = k\*S(n, k) + S(n, k-1)

**How does above recursive formula work?**

When we add a (n+1)’th element to k partitions, there are two possibilities.

1) It is added as a single element set to existing partitions, i.e, S(n, k-1)

2) It is added to all sets of every partition, i.e., k\*S(n, k)

S(n, k) is called [Stirling numbers of the second kind](https://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind)

First few Bell numbers are 1, 1, 2, 5, 15, 52, 203, ….

A **Simple Method** to compute n’th Bell Number is to one by one compute S(n, k) for k = 1 to n and return sum of all computed values. Refer [this](https://www.geeksforgeeks.org/count-number-of-ways-to-partition-a-set-into-k-subsets/) for computation of S(n, k).

Below is Dynamic Programming based implementation of the above recursive code using the Stirling number-

# python program to find number of ways of partitioning it.

n **=** 5

s **=** [[0 **for** \_ **in** range(n**+**1)] **for** \_ **in** range(n**+**1)]

**for** i **in** range(n**+**1):

**for** j **in** range(n**+**1):

**if** j > i:

**continue**

**elif**(i**==**j):

            s[i][j] **=** 1

**elif**(i**==**0 **or** j**==**0):

            s[i][j]**=**0

**else**:

            s[i][j] **=** j**\***s[i**-**1][j] **+** s[i**-**1][j**-**1]

ans **=** 0

**for** i **in** range(0,n**+**1):

    ans**+=**s[n][i]

print(ans)

**Output**

52

**Time complexity:** O(N2)

**Auxiliary Space:**O(N2)

A **Better Method** is to use [Bell Triangle](https://en.wikipedia.org/wiki/Bell_triangle). Below is a sample Bell Triangle for first few Bell Numbers.

1  
1 2  
2 3 5  
5 7 10 15  
15 20 27 37 52

The triangle is constructed using below formula.

// If this is first column of current row 'i'  
If j == 0  
 // Then copy last entry of previous row  
 // Note that i'th row has i entries  
 Bell(i, j) = Bell(i-1, i-1)

// If this is not first column of current row  
Else   
 // Then this element is sum of previous element   
 // in current row and the element just above the  
 // previous element  
 Bell(i, j) = Bell(i-1, j-1) + Bell(i, j-1)

**Interpretation:**

Then Bell(n, k) counts the number of partitions of the set {1, 2, …, n + 1} in which the element k + 1 is the largest element that can be alone in its set.

For example, Bell(3, 2) is 3, it is count of number of partitions of {1, 2, 3, 4} in which 3 is the largest singleton element. There are three such partitions:

{1}, {2, 4}, {3}  
 {1, 4}, {2}, {3}  
 {1, 2, 4}, {3}.

Below is Dynamic Programming based implementation of above recursive formula.

# A Python program to find n'th Bell number

**def** bellNumber(n):

    bell **=** [[0 **for** i **in** range(n**+**1)] **for** j **in** range(n**+**1)]

    bell[0][0] **=** 1

**for** i **in** range(1, n**+**1):

        # Explicitly fill for j = 0

        bell[i][0] **=** bell[i**-**1][i**-**1]

        # Fill for remaining values of j

**for** j **in** range(1, i**+**1):

            bell[i][j] **=** bell[i**-**1][j**-**1] **+** bell[i][j**-**1]

**return** bell[n][0]

# Driver program

**for** n **in** range(6):

**print**('Bell Number', n, 'is', bellNumber(n))

# This code is contributed by Soumen Ghosh

**Output**

Bell Number 0 is 1  
Bell Number 1 is 1  
Bell Number 2 is 2  
Bell Number 3 is 5  
Bell Number 4 is 15  
Bell Number 5 is 52

**Time Complexity:** O(N2)

**Auxiliary Space:**O(N2)

We will soon be discussing other more efficient methods of computing Bell Numbers.

**Another problem that can be solved by Bell Numbers**.

A number is [squarefree](https://en.wikipedia.org/wiki/Square-free_integer) if it is not divisible by a perfect square other than 1. For example, 6 is a square free number but 12 is not as it is divisible by 4.

Given a squarefree number x, find the number of different multiplicative partitions of x. The number of multiplicative partitions is Bell(n) where n is number of prime factors of x. For example x = 30, there are 3 prime factors of 2, 3 and 5. So the answer is Bell(3) which is 5. The 5 partitions are 1 x 30, 2 x15, 3 x 10, 5 x 6 and 2 x 3 x 5.

**Exercise:**

The above implementation causes arithmetic overflow for slightly larger values of n. Extend the above program so that results are computed under modulo 1000000007 to avoid overflows.

**Reference:**

<https://en.wikipedia.org/wiki/Bell_number>

<https://en.wikipedia.org/wiki/Bell_triangle>

**10.Binomial Coefficient | DP-9**

The following are the common definitions of [Binomial Coefficients](http://en.wikipedia.org/wiki/Binomial_coefficient).

*A*[*binomial coefficient*](http://en.wikipedia.org/wiki/Binomial_coefficient)*C(n, k) can be defined as the coefficient of x^k in the expansion of (1 + x)^n.*

*A binomial coefficient C(n, k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (or k-combinations) of a n-element set.*

**The Problem**

*Write a function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k).* For example, your function should return 6 for n = 4 and k = 2, and it should return **10**for **n = 5**and **k = 2**.

nCr

**1) Optimal Substructure**

The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k)  
 C(n, 0) = C(n, n) = 1

Following is a simple recursive implementation that simply follows the recursive structure mentioned above.

# A naive recursive Python implementation

**def** binomialCoeff(n, k):

**if** k > n:

**return** 0

**if** k **==** 0 **or** k **==** n:

**return** 1

    # Recursive Call

**return** binomialCoeff(n**-**1, k**-**1) **+** binomialCoeff(n**-**1, k)

# Driver Program to test ht above function

n **=** 5

k **=** 2

print ("Value of C(%d,%d) is (%d)" **%** (n, k,

                                     binomialCoeff(n, k)))

# This code is contributed by Nikhil Kumar Singh (nickzuck\_007)

**Output**

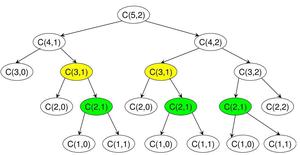
Value of C(5, 2) is 10

***Time Complexity:***O(n\*max(k,n-k))

***Auxiliary Space:***O(n\*max(k,n-k))

**2) Overlapping Subproblems**

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.



*Binomial Coefficients Recursion tree for C(5,2)*

Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), re-computations of the same subproblems can be avoided by constructing a temporary 2D-array C[][] in a bottom-up manner. Following is Dynamic Programming-based implementation.

# A Dynamic Programming based Python

# Program that uses table C[][]

# to calculate the Binomial Coefficient

# Returns value of Binomial Coefficient C(n, k)

**def** binomialCoef(n, k):

    C **=** [[0 **for** x **in** range(k**+**1)] **for** x **in** range(n**+**1)]

    # Calculate value of Binomial

    # Coefficient in bottom up manner

**for** i **in** range(n**+**1):

**for** j **in** range(min(i, k)**+**1):

            # Base Cases

**if** j **==** 0 **or** j **==** i:

                C[i][j] **=** 1

            # Calculate value using

            # previously stored values

**else**:

                C[i][j] **=** C[i**-**1][j**-**1] **+** C[i**-**1][j]

**return** C[n][k]

# Driver program to test above function

n **=** 5

k **=** 2

**print**("Value of C[" **+** str(n) **+** "][" **+** str(k) **+** "] is "

**+** str(binomialCoef(n, k)))

# This code is contributed by Bhavya Jain

**Output**

Value of C[5][2] is 10

**Time Complexity:** O(n\*k)

**Auxiliary Space:** O(n\*k)

Following is a space-optimized version of the above code. The following code only uses O(k). Thanks to **AK** for suggesting this method.

# Python program for Optimized

# Dynamic Programming solution to

# Binomial Coefficient. This one

# uses the concept of pascal

# Triangle and less memory

**def** binomialCoeff(n, k):

    # Declaring an empty array

    C **=** [0 **for** i **in** range(k**+**1)]

    C[0] **=** 1  # since nC0 is 1

**for** i **in** range(1, n**+**1):

        # Compute next row of pascal triangle using

        # the previous row

        j **=** min(i, k)

**while** (j > 0):

            C[j] **=** C[j] **+** C[j**-**1]

            j **-=** 1

**return** C[k]

# Driver Code

n **=** 5

k **=** 2

print ("Value of C(%d,%d) is %d" **%** (n, k, binomialCoeff(n, k)))

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

Value of C(5, 2) is 10

**Time Complexity:** O(n\*k)

**Auxiliary Space:** O(k)

**Explanation:**

1==========>> n = 0, C(0,0) = 1

1–1========>> n = 1, C(1,0) = 1, C(1,1) = 1

1–2–1======>> n = 2, C(2,0) = 1, C(2,1) = 2, C(2,2) = 1

1–3–3–1====>> n = 3, C(3,0) = 1, C(3,1) = 3, C(3,2) = 3, C(3,3)=1

1–4–6–4–1==>> n = 4, C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3)=4, C(4,4)=1

So here every loop on i, builds i’th row of pascal triangle, using (i-1)th row

At any time, every element of array C will have some value (ZERO or more) and in the next iteration, the value for those elements comes from the previous iteration.

In statement,

C[j] = C[j] + C[j-1]

The right-hand side represents the value coming from the previous iteration (A row of Pascal’s triangle depends on the previous row). The left-Hand side represents the value of the current iteration which will be obtained by this statement.

Let's say we want to calculate C(4, 3),   
i.e. n=4, k=3:

All elements of array C of size 4 (k+1) are  
initialized to ZERO.

i.e. C[0] = C[1] = C[2] = C[3] = C[4] = 0;  
Then C[0] is set to 1

For i = 1:  
C[1] = C[1] + C[0] = 0 + 1 = 1 ==>> C(1,1) = 1

For i = 2:  
C[2] = C[2] + C[1] = 0 + 1 = 1 ==>> C(2,2) = 1  
C[1] = C[1] + C[0] = 1 + 1 = 2 ==>> C(2,1) = 2

For i=3:  
C[3] = C[3] + C[2] = 0 + 1 = 1 ==>> C(3,3) = 1  
C[2] = C[2] + C[1] = 1 + 2 = 3 ==>> C(3,2) = 3  
C[1] = C[1] + C[0] = 2 + 1 = 3 ==>> C(3,1) = 3

For i=4:  
C[4] = C[4] + C[3] = 0 + 1 = 1 ==>> C(4,4) = 1  
C[3] = C[3] + C[2] = 1 + 3 = 4 ==>> C(4,3) = 4  
C[2] = C[2] + C[1] = 3 + 3 = 6 ==>> C(4,2) = 6  
C[1] = C[1] + C[0] = 3 + 1 = 4 ==>> C(4,1) = 4

C(4,3) = 4 is would be the answer in our example.

**Memoization Approach:**The idea is to create a lookup table and follow the recursive top-down approach. Before computing any value, we check if it is already in the lookup table. If yes, we return the value. Else we compute the value and store it in the lookup table. Following is the Top-down approach of dynamic programming to finding the value of the Binomial Coefficient.

# A Dynamic Programming based solution

# that uses table dp[][] to calculate

# the Binomial Coefficient. A naive

# recursive approach with table

# Python3 implementation

# Returns value of Binomial

# Coefficient C(n, k)

**def** binomialCoeffUtil(n, k, dp):

    # If value in lookup table then return

**if** dp[n][k] !**= -**1:

**return** dp[n][k]

    # Store value in a table before return

**if** k **==** 0:

        dp[n][k] **=** 1

**return** dp[n][k]

    # Store value in table before return

**if** k **==** n:

        dp[n][k] **=** 1

**return** dp[n][k]

    # Save value in lookup table before return

    dp[n][k] **=** (binomialCoeffUtil(n **-** 1, k **-** 1, dp) **+**

                binomialCoeffUtil(n **-** 1, k, dp))

**return** dp[n][k]

**def** binomialCoeff(n, k):

    # Make a temporary lookup table

    dp **=** [ [ **-**1 **for** y **in** range(k **+** 1) ]

**for** x **in** range(n **+** 1) ]

**return** binomialCoeffUtil(n, k, dp)

# Driver code

n **=** 5

k **=** 2

**print**("Value of C(" **+** str(n) **+**

               ", " **+** str(k) **+** ") is",

               binomialCoeff(n, k))

# This is code is contributed by Prateek Gupta

***Time Complexity:***O(n\*k)

***Auxiliary Space:***O(n\*k)

**Output**

Value of C(5, 2) is 10

**11.Cancellation of factors between numerator and denominator:**

nCr = (n-r+1)\*(n-r+2)\*….\*n / (r!)

Create an array arr of numbers from n-r+1 to n which will be of size r. As nCr is always an integer, all numbers in the denominator should cancel with the product of the numerator (represented by arr).

for i = 1 to i = r,

        search arr, if arr[j] and i have gcd>1, divide both by the gcd and when i becomes 1, stop the search

Now, the answer is just the product of arr, whose value mod 10^9+7 can be found using a single pass and the formula use (a\*b)%mod = (a%mod \* b%mod)%mod

**import** math

**class** GFG:

**def** nCr(self, n, r):

**def** gcd(a,b): # function to find gcd of two numbers in O(log(min(a,b)))

**if** b**==**0: # base case

**return** a

**return** gcd(b,a**%**b)

**if** r>n:

**return** 0

**if** r>n**-**r: # C(n,r) = C(n,n-r) better time complexity for lesser r value

            r **=** n**-**r

        mod **=** 10**\*\***9 **+** 7

        arr **=** list(range(n**-**r**+**1,n**+**1)) # array of elements from n-r+1 to n

        ans **=** 1

**for** i **in** range(1,r**+**1): # for numbers from 1 to r find arr[j] such that gcd(i,arr[j])>1

            j**=**0

**while** j<len(arr):

                x **=** gcd(i,arr[j])

**if** x>1:

                    arr[j] **//=** x # if gcd>1, divide both by gcd

                    i **//=** x

**if** arr[j]**==**1: # if element becomes 1, its of no use anymore so delete from arr

**del** arr[j]

                    j **-=** 1

**if** i**==**1:

**break** # if i becomes 1, no need to search arr

                j **+=** 1

**for** i **in** arr: # single pass to multiply the numerator

            ans **=** (ans**\***i)**%**mod

**return** ans

     # Driver code

n **=** 5

k **=** 2

ob **=** GFG()

**print**("Value of C(" **+** str(n) **+**

               ", " **+** str(k) **+** ") is",

               ob.nCr(n, k))

# This is code is contributed by Gautam Wadhwani

**Output**

Value of C(5, 2) is 10

**Time Complexity:** O(( min(r, n-r)^2 ) \* log(n)),   **useful when n >> r  or  n >> (n-r)**

**Auxiliary Space:** O(min(r, n-r))

See this for [GCD in logarithmic time](https://www.geeksforgeeks.org/c-program-find-gcd-hcf-two-numbers/)

**12.Prime factorization of every number from 1 to n using Sieve of Eratosthenes :**

1. Create an array SPF of size n+1 to the smallest prime factor of each number from 1 to n

Set SPF[i] = i for all i = 1 to i = n

2. Use Sieve of Eratosthenes:

for i = 2 to i = n:  
 if i is prime,  
 for all multiples j of i, j<=n:  
 if SPF[j] equals j, set SPF[j] = i

3. Once, we know the SPF of each number from 1 to n, we can find the prime factorization of any number from 1 to n in O(log(n)) time using recursive division by SPF until the number becomes 1

Now, nCr = (n-r+1)\*(r+2)\* ... \*(n) / (r)!

4. Create a dictionary (or hashmap) to store the frequency of each prime in the prime factorization of the actual value of nCr.

5. So, just calculate the frequency of each prime in nCr and multiply them raised to the power of their frequency.

6. For the numerator, iterate through i = n-r+1 to i = n, and for all prime factors of i, store their frequency in a dictionary.

( prime\_pow[prime\_factor] += freq\_in\_i )

7. For the denominator, iterate through i = 1 to i = r and now subtract the frequency instead of adding.

8. Now, traverse the dictionary and multiply the answer to (prime ^ prime\_pow[prime]) % (10^9 + 7)

ans = (ans \* pow(prime, prime\_pow[prime], mod) ) % mod

# Python code for the above approach

**import** math

**class** GFG:

**def** nCr(self, n, r):

        # Base case

**if** r > n:

**return** 0

        # C(n,r) = C(n,n-r) Complexity for this

        # code is lesser for lower n-r

**if** n **-** r > r:

            r **=** n **-** r

        # List to store smallest prime factor

        # of every number from 1 to n

        SPF **=** [i **for** i **in** range(n**+**1)]

**for** i **in** range(4, n**+**1, 2):

            # set smallest prime factor of

            # all even numbers as 2

            SPF[i] **=** 2

**for** i **in** range(3, n**+**1, 2):

**if** i**\***i > n:

**break**

            # Check if i is prime

**if** SPF[i] **==** i:

                # All multiples of i are composite

                # (and divisible by i) so add i to

                # their prime factorization getpow(j,i) times

**for** j **in** range(i**\***i, n**+**1, i):

**if** SPF[j] **==** j:

                        # set smallest prime factor

                        # of j to i only if it is

                        # not previously set

                        SPF[j] **=** i

         # dictionary to store power of each prime in C(n,r)

        prime\_pow **=** {}

        # For numerator count frequency

        # of each prime factor

**for** i **in** range(r**+**1, n**+**1):

            t **=** i

            # Recursive division to

            # find prime factorization of i

**while** t > 1:

**if not** SPF[t] **in** prime\_pow:

                    prime\_pow[SPF[t]] **=** 1

**else**:

                    prime\_pow[SPF[t]] **+=** 1

                t **//=** SPF[t]

        # For denominator subtract the

        # power of each prime factor

**for** i **in** range(1, n**-**r**+**1):

            t **=** i

            # Recursive division to

            # find prime factorization of i

**while** t > 1:

                prime\_pow[SPF[t]] **-=** 1

                t **//=** SPF[t]

        ans **=** 1

        mod **=** 10**\*\***9 **+** 7

         # Use (a\*b)%mod = (a%mod \* b%mod)%mod

**for** i **in** prime\_pow:

            # pow(base,exp,mod) is used to

            # find (base^exp)%mod fast

            ans **=** (ans**\***pow(i, prime\_pow[i], mod)) **%** mod

**return** ans

# Driver code

n **=** 5

k **=** 2

ob **=** GFG()

**print**("Value of C(" **+** str(n) **+**

      ", " **+** str(k) **+** ") is",

      ob.nCr(n, k))

# This is code is contributed by Gautam Wadhwani

**Output**

Value of C(5, 2) is 10

**Time Complexity:** O(n\*log(n)),  **so useful when r->n/2**

**Auxiliary Space:** O(n)

See this for [Prime factorization in O(log(n))](https://www.geeksforgeeks.org/prime-factorization-using-sieve-olog-n-multiple-queries/)

**Another Approach: (Modular Inversion technique)**

1. The general formula of nCr is ( n\*(n-1)\*(n-2)\* … \*(n-r+1) ) / (r!). We can directly use this formula to find nCr. But that will overflow out of bound. We need to find nCr mod m so that it doesn’t overflow. We can easily do it with modular arithmetic formula.

for the n\*(n-1)\*(n-2)\* ... \*(n-r+1) part we can use the formula,  
(a\*b) mod m = ((a % m) \* (b % m)) % m

2. and for the 1/r! part, we need to find the modular inverse of every number from 1 to r. Then use the same formula above with a modular inverse of 1 to r. We can find modular inverse in O(r) time using  the formula,

inv[1] = 1  
inv[i] = − ⌊m/i⌋ \* inv[m mod i] mod m  
To use this formula, m has to be a prime.

In the practice problem, we need to show the answer with modulo 1000000007 which is a prime.

So, this technique will work.

# Python3 program for the above approach

# Function to find binomial

# coefficient

**def** binomialCoeff(n, r):

**if** (r > n):

**return** 0

    m **=** 1000000007

    inv **=** [0 **for** i **in** range(r **+** 1)]

    inv[0] **=** 1;

**if**(r**+**1>**=**2)

    inv[1] **=** 1;

    # Getting the modular inversion

    # for all the numbers

    # from 2 to r with respect to m

    # here m = 1000000007

**for** i **in** range(2, r **+** 1):

        inv[i] **=** m **-** (m **//** i) **\*** inv[m **%** i] **%** m

    ans **=** 1

    # for 1/(r!) part

**for** i **in** range(2, r **+** 1):

        ans **=** ((ans **%** m) **\*** (inv[i] **%** m)) **%** m

    # for (n)\*(n-1)\*(n-2)\*...\*(n-r+1) part

**for** i **in** range(n, n **-** r, **-**1):

        ans **=** ((ans **%** m) **\*** (i **%** m)) **%** m

**return** ans

# Driver code

n **=** 5

r **=** 2

print("Value of C(" ,n , ", " , r ,

      ") is ",binomialCoeff(n, r))

# This code is contributed by rohan07

**Output**

Value of C(5, 2) is 10

**Time Complexity:** O(n+k)

**Auxiliary Space:** O(k)

**13.Coin Change | DP-7**

Given an integer array of **coins[ ]**of size**N**representing different types of currency and an integer **sum**, The task is to find the number of ways to make **sum** by using different combinations from**coins[]**.

**Note:**Assume that you have an infinite supply of each type of coin.

**Examples:**

***Input:****sum = 4, coins[] = {1,2,3},*

***Output:****4*

***Explanation:****there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}.*

***Input:****sum = 10, coins[] = {2, 5, 3, 6}*

***Output:****5*

***Explanation:****There are five solutions:*

*{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.*

Coin Change

**Coin Change Problem using**[**Recursion**](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)**:**

*Solve the Coin Change is to traverse the array by applying the recursive solution and keep finding the possible ways to find the occurrence.*

**Illustration:**

*It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for coins[] = {1, 2, 3} and n = 5.*

*The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many****subproblems****being called more than once.*

*C() –> count()*

*C({1,2,3}, 5)*

*/                     \*

*/                         \*

*C({1,2,3}, 2)                   C({1,2}, 5)*

*/       \                          /    \*

*/              \                   /            \*

*C({1,2,3}, -1)  C({1,2}, 2)        C({1,2}, 3)    C({1}, 5)*

*/    \                  /     \           /     \*

*/       \                /       \         /        \*

*C({1,2},0)  C({1},2)   C({1,2},1) C({1},3)    C({1}, 4)  C({}, 5)*

*/ \           / \          /\           /     \*

*/      \      /     \       /   \        /        \*

*.      .  .     .   .     .              C({1}, 3)      C({}, 4)*

*/ \*

*/   \                                                 .*

Follow the below steps to Implement the idea:

* We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.
* If we are at coins[n-1], we can take as many instances of that coin ( unbounded inclusion ) i.e **count(coins, n, sum – coins[n-1] )**; then we move to coins[n-2].
* After moving to coins[n-2], we can’t move back and can’t make choices for coins[n-1] i.e **count(coins, n-1, sum)**.
* Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(coins, n, sum – coins[n-1] ) + count(coins, n-1, sum );**

Below is the Implementation of the above approach.

# Recursive Python3 program for

# coin change problem.

# Returns the count of ways we can sum

# coins[0...n-1] coins to get sum "sum"

**def** count(coins, n, sum):

    # If sum is 0 then there is 1

    # solution (do not include any coin)

**if** (sum **==** 0):

**return** 1

    # If sum is less than 0 then no

    # solution exists

**if** (sum < 0):

**return** 0

    # If there are no coins and sum

    # is greater than 0, then no

    # solution exist

**if** (n <**=** 0):

**return** 0

    # count is sum of solutions (i)

    # including coins[n-1] (ii) excluding coins[n-1]

**return** count(coins, n **-** 1, sum) **+** count(coins, n, sum**-**coins[n**-**1])

# Driver program to test above function

coins **=** [1, 2, 3]

n **=** len(coins)

print(count(coins, n, 4))

# This code is contributed by Smitha Dinesh Semwal

**Output**

4

**Time Complexity:**O(2sum)

**Auxiliary Space:**O(target)

Since the same sub-problems are called again, this problem has the Overlapping Subproblems property. So the Coin Change problem has both properties (see [this](https://www.geeksforgeeks.org/archives/12635)and [this](https://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of the same subproblems can be avoided by constructing a temporary array table[][] in a bottom-up manner.

**14.Coin Change By Using**[**Dynamic Programming:**](https://www.geeksforgeeks.org/dynamic-programming/)

*The Idea to Solve this Problem is by using the Bottom Up Memoization. Here is the Bottom up approach to solve this Problem.*

Follow the below steps to Implement the idea:

* Using 2-D vector to store the Overlapping subproblems.
* Traversing the whole array to find the solution and storing in the memoization table.
* Using the memoization table to find the optimal solution.

Below is the implementation of the above Idea.

# Dynamic Programming Python implementation of Coin

# Change problem

**def** count(coins, n, sum):

    # We need sum+1 rows as the table is constructed

    # in bottom up manner using the base case 0 value

    # case (sum = 0)

    table **=** [[0 **for** x **in** range(n)] **for** x **in** range(sum**+**1)]

    # Fill the entries for 0 value case (n = 0)

**for** i **in** range(n):

        table[0][i] **=** 1

    # Fill rest of the table entries in bottom up manner

**for** i **in** range(1, sum**+**1):

**for** j **in** range(n):

            # Count of solutions including coins[j]

            x **=** table[i **-** coins[j]][j] **if** i**-**coins[j] >**=** 0 **else** 0

            # Count of solutions excluding coins[j]

            y **=** table[i][j**-**1] **if** j >**=** 1 **else** 0

            # total count

            table[i][j] **=** x **+** y

**return** table[sum][n**-**1]

# Driver program to test above function

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    coins **=** [1, 2, 3]

    n **=** len(coins)

    sum **=** 4

    print(count(coins, n, sum))

# This code is contributed by Bhavya Jain

**Output**

4

**Time Complexity**: O(M\*sum)

**Auxiliary Space**: O(M\*sum)

**15.Coin change Using the Space Optimized 1D array:**

*The Idea to**Solve this Problem is by using the Bottom Up(Tabulation). By using the linear array for space optimization.*

Follow the below steps to Implement the idea:

* Initialize with a linear array **table** with values equal to 0.
* With **sum = 0**, there is a way.
* Update the level wise number of ways of coin till the **ith** coin.
* Solve till **j <= sum.**

Below is the implementation of the above Idea.

# Dynamic Programming Python implementation of Coin

# Change problem

**def** count(coins, n, sum):

    # table[i] will be storing the number of solutions for

    # value i. We need sum+1 rows as the table is constructed

    # in bottom up manner using the base case (sum = 0)

    # Initialize all table values as 0

    table **=** [0 **for** k **in** range(sum**+**1)]

    # Base case (If given value is 0)

    table[0] **=** 1

    # Pick all coins one by one and update the table[] values

    # after the index greater than or equal to the value of the

    # picked coin

**for** i **in** range(0, n):

**for** j **in** range(coins[i], sum**+**1):

            table[j] **+=** table[j**-**coins[i]]

**return** table[sum]

# Driver program to test above function

coins **=** [1, 2, 3]

n **=** len(coins)

sum **=** 4

x **=** count(coins, n, sum)

print(x)

# This code is contributed by Afzal Ansari

**Output**

4

**Time Complexity**: O(N\*sum)

**Auxiliary Space**: O(sum)

**16.Coin change using the**[**Top Down (Memoization) Dynamic Programming:**](https://www.geeksforgeeks.org/memoization-1d-2d-and-3d/)

*The idea is to find the Number of ways of Denominations By using the Top Down (Memoization).*

Follow the below steps to Implement the idea:

* Creating a 2-D vector to store the Overlapping Solutions
* Keep Track of the overlapping subproblems while Traversing the array **coins[]**
* Recall them whenever needed

Below is the implementation using the Top Down Memoized Approach

# Python program for the above approach

**def** coinchange(a, v, n, dp):

**if** (v **==** 0):

        dp[n][v] **=** 1

**return** dp[n][v]

**if** (n **==** 0):

**return** 0

**if** (dp[n][v] !**= -**1):

**return** dp[n][v]

**if** (a[n **-** 1] <**=** v):

        # Either Pick this coin or not

        dp[n][v] **=** coinchange(a, v **-** a[n **-** 1], n, dp) **+** \

            coinchange(a, v, n **-** 1, dp)

**return** dp[n][v]

**else**:  # We have no option but to leave this coin

        dp[n][v] **=** coinchange(a, v, n **-** 1, dp)

**return** dp[n][v]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    tc **=** 1

**while** (tc !**=** 0):

        n **=** 3

        v **=** 4

        a **=** [1, 2, 3]

        dp **=** [[**-**1 **for** i **in** range(v**+**1)] **for** j **in** range(n**+**1)]

        res **=** coinchange(a, v, n, dp)

**print**(res)

        tc **-=** 1

# This code is contributed by Rajput-Ji

**Output**

4

**Time Complexity:** O(N\*sum)

**Auxiliary Space:** O(N\*sum)

**17.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.

Subset Sum Problem

**Method 1:** Recursion.

**Approach:** For the recursive approach we will consider two cases.

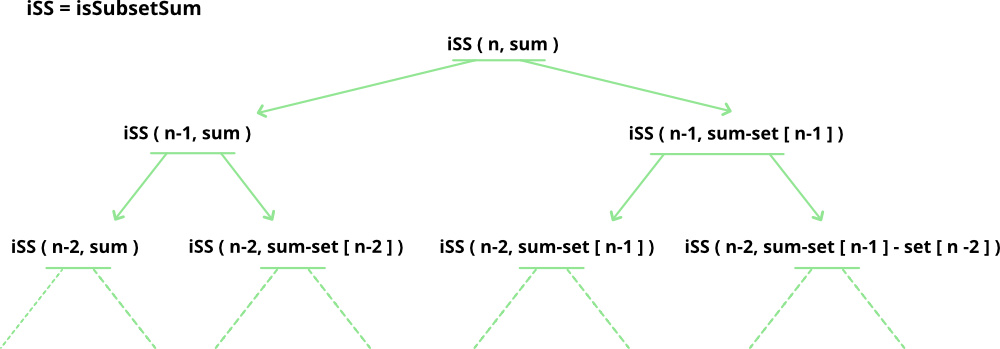
1. Consider the last element and now the **required sum = target sum – value of ‘last’ element** and **number of elements = total elements – 1**
2. Leave the ‘last’ element and now the **required sum = target sum**and **number of elements = total elements – 1**

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}



# A recursive solution for subset sum

# problem

# Returns true if there is a subset

# of set[] with sun equal to given sum

**def** isSubsetSum(set, n, 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) **or** isSubsetSum(

        set, n**-**1, sum**-**set[n**-**1])

# Driver code

set **=** [3, 34, 4, 12, 5, 2]

sum **=** 9

n **=** len(set)

**if** (isSubsetSum(set, n, sum) **==** True):

    print("Found a subset with given sum")

**else**:

    print("No subset with given sum")

# This code is contributed by Nikita Tiwari.

**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](http://en.wikipedia.org/wiki/NP-complete) (*There is no known polynomial time solution for this problem*).

**Space Complexity:** O(n) where n is recursion stack space.

**Method 2:** To solve the problem in [Pseudo-polynomial time](http://en.wikipedia.org/wiki/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:

# A Dynamic Programming solution for subset

# sum problem Returns true if there is a subset of

# set[] with sun equal to given sum

# Returns true if there is a subset of set[]

# with sum equal to given sum

**def** isSubsetSum(set, n, 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

    subset **=**([[False **for** i **in** range(sum **+** 1)]

**for** i **in** range(n **+** 1)])

    # If sum is 0, then answer is true

**for** i **in** range(n **+** 1):

        subset[i][0] **=** True

    # If sum is not 0 and set is empty,

    # then answer is false

**for** i **in** range(1, sum **+** 1):

         subset[0][i]**=** False

    # Fill the subset table in bottom up manner

**for** i **in** range(1, n **+** 1):

**for** j **in** range(1, sum **+** 1):

**if** j<set[i**-**1]:

                subset[i][j] **=** subset[i**-**1][j]

**if** j>**=** set[i**-**1]:

                subset[i][j] **=** (subset[i**-**1][j] **or**

                                subset[i **-** 1][j**-**set[i**-**1]])

    # uncomment this code to print table

    # for i in range(n + 1):

    # for j in range(sum + 1):

    # print (subset[i][j], end =" ")

    # print()

**return** subset[n][sum]

# Driver code

**if** \_\_name\_\_**==**'\_\_main\_\_':

    set **=** [3, 34, 4, 12, 5, 2]

    sum **=** 9

    n **=** len(set)

**if** (isSubsetSum(set, n, sum) **==** True):

        print("Found a subset with given sum")

**else**:

        print("No subset with given sum")

# This code is contributed by

# sahil shelangia.

**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:

# Python program for the above approach

# Taking the matrix as globally

tab **=** [[**-**1 **for** i **in** range(2000)] **for** j **in** range(2000)]

# Check if possible subset with

# given sum is possible or not

**def** subsetSum(a, n, sum):

    # 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):

        tab[n **-** 1][sum] **=** subsetSum(a, n **-** 1, sum)

**return** tab[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

        tab[n **-** 1][sum] **=** subsetSum(a, n **-** 1, sum)

**return** tab[n **-** 1][sum] **or** subsetSum(a, n **-** 1, sum **-** a[n **-** 1])

# Driver Code

n **=** 5

a **=** [1, 5, 3, 7, 4]

sum **=** 12

**if** (subsetSum(a, n, sum)):

    print("YES")

**else**:

    print("NO")

# This code is contributed by shivani.

**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.

**18.Introduction and Dynamic Programming solution to compute nCr%p**

Given three numbers n, r and p, compute value of nCr mod p.

**Example:**

**Input:**  n = 10, r = 2, p = 13  
**Output:** 6  
**Explanation:** 10C2 is 45 and 45 % 13 is 6.

[We strongly recommend that you click here and practice it, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/ncr1019/1)

**METHOD 1: (Using Dynamic Programming)**

A **Simple Solution** is to first compute nCr, then compute nCr % p. This solution works fine when the value of nCr is small.

**What if the value of nCr is large?**

The value of nCr%p is generally needed for large values of n when nCr cannot fit in a variable, and causes overflow. So computing nCr and then using modular operator is not a good idea as there will be overflow even for slightly larger values of n and r. For example the methods discussed [here](https://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/) and [here](https://www.geeksforgeeks.org/space-and-time-efficient-binomial-coefficient/) cause overflow for n = 50 and r = 40.

The idea is to compute nCr using below formula

C(n, r) = C(n-1, r-1) + C(n-1, r)  
 C(n, 0) = C(n, n) = 1

**Working of Above formula and Pascal Triangle:**

Let us see how above formula works for C(4, 3)

1==========>> n = 0, C(0, 0) = 1

1–1========>> n = 1, C(1, 0) = 1, C(1, 1) = 1

1–2–1======>> n = 2, C(2, 0) = 1, C(2, 1) = 2, C(2, 2) = 1

1–3–3–1====>> n = 3, C(3, 0) = 1, C(3, 1) = 3, C(3, 2) = 3, C(3, 3)=1

1–4–6–4–1==>> n = 4, C(4, 0) = 1, C(4, 1) = 4, C(4, 2) = 6, C(4, 3)=4, C(4, 4)=1

So here every loop on i, builds i’th row of pascal triangle, using (i-1)th row

**Extension of above formula for modular arithmetic:**

We can use distributive property of modular operator to find nCr % p using above formula.

C(n, r)%p = [ C(n-1, r-1)%p + C(n-1, r)%p ] % p  
 C(n, 0) = C(n, n) = 1

The above formula can be implemented using Dynamic Programming using a 2D array.

The 2D array based dynamic programming solution can be further optimized by constructing one row at a time. See Space optimized version in below post for details.

[Binomial Coefficient using Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/)

Below is implementation based on the space optimized version discussed in above post.

# A Dynamic Programming based solution to compute nCr % p

# Returns nCr % p

**def** nCrModp(n, r, p):

    # Optimization for the cases when r is large

    # compared to n-r

**if** (r > n**-** r):

        r **=** n **-** r

    # The array C is going to store last row of

    # pascal triangle at the end. And last entry

    # of last row is nCr.

    C **=** [0 **for** i **in** range(r **+** 1)]

    C[0] **=** 1 # Top row of Pascal Triangle

    # One by constructs remaining rows of Pascal

    # Triangle from top to bottom

**for** i **in** range(1, n **+** 1):

        # Fill entries of current row

        # using previous row values

**for** j **in** range(min(i, r), 0, **-**1):

            # nCj = (n - 1)Cj + (n - 1)C(j - 1)

            C[j] **=** (C[j] **+** C[j**-**1]) **%** p

**return** C[r]

# Driver Program

n **=** 10

r **=** 2

p **=** 13

print('Value of nCr % p is', nCrModp(n, r, p))

# This code is contributed by Soumen Ghosh

**Output**

Value of nCr % p is 6

Time complexity of above solution is O(n\*r) and it requires O(r) space. There are more and better solutions to above problem.

[Compute nCr % p | Set 2 (Lucas Theorem)](https://www.geeksforgeeks.org/compute-ncr-p-set-2-lucas-theorem/)

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

**METHOD 2(Using Pascal Triangle and Dynamic Pro)**

Another approach lies in utilizing the concept of the Pascal Triangle. Instead of calculating the nCr value for every n starting from n=0 till n=n, the approach aims at using the nth row itself for the calculation. The method proceeds by finding out a general relationship between nCr and nCr-1.

***FORMULA: C(n,r)=C(n,r-1)\* (n-r+1)/r***

**Example:**

For instance, take n=5 and r=3.

**Input:** n = 5, r = 3, p = 1000000007  
**Output:** 6  
**Explanation:** 5C3 is 10 and 10 % 100000007 is 10.

As per the formula,  
C(5,3)=5!/(3!)\*(2!)  
C(5,3)=10

Also,  
C(5,2)=5!/(2!)\*(3!)  
C(5,2)=10

**Let's try applying the above formula.**

C(n,r)=C(n,r-1)\* (n-r+1)/r  
C(5,3)=C(5,2)\*(5-3+1)/3  
C(5,3)=C(5,2)\*1  
C(5,3)=10\*1

The above example shows that C(n,r) can be easily calculated by calculating C(n,r-1) and multiplying the result with the term (n-r+1)/r. But this multiplication may cause integer overflow for large values of n. To tackle this situation, use modulo multiplication, and modulo division concepts in order to achieve optimizations in terms of integer overflow.

**Let’s find out how to build Pascal Triangle for the same.**

1D array declaration can be further optimized by just the declaration of a single variable to perform calculations. However, integer overflow demands other functions too for the final implementation.

The post below mentions the **space and time-optimized** implementation for the binary coefficient calculation.

# Python3 program to find the nCr%p

# based on optimal Dynamic

# Programming implementation and

# Pascal Triangle concepts

# Returns (a \* b) % mod

**def** moduloMultiplication(a, b, mod):

    # Initialize result

    res **=** 0

    # Update a if it is more than

    # or equal to mod

    a **%=** mod

**while** (b):

        # If b is odd, add a with result

**if** (b & 1):

            res **=** (res **+** a) **%** mod

        # Here we assume that doing 2\*a

        # doesn't cause overflow

        a **=** (2 **\*** a) **%** mod

        b >>**=** 1    # b = b / 2

**return** res

# Global Variables

x, y **=** 0, 1

# Function for extended Euclidean Algorithm

**def** gcdExtended(a, b):

**global** x, y

    # Base Case

**if** (a **==** 0):

        x **=** 0

        y **=** 1

**return** b

    # To store results of recursive call

    gcd **=** gcdExtended(b **%** a, a)

    x1 **=** x

    y1 **=** y

    # Update x and y using results of recursive

    # call

    x **=** y1 **-** int(b **/** a) **\*** x1

    y **=** x1

**return** gcd

**def** modInverse(a, m):

    g **=** gcdExtended(a, m)

    # Return -1 if b and m are not co-prime

**if** (g !**=** 1):

**return -**1

    # m is added to handle negative x

**return** (x **%** m **+** m) **%** m

# Function to compute a/b under modulo m

**def** modDivide(a, b, m):

    a **=** a **%** m

    inv **=** modInverse(b, m)

**if** (inv **== -**1):

**return** 0

**else**:

**return** (inv **\*** a) **%** m

# Function to calculate nCr % p

**def** nCr(n, r, p):

    # Edge Case which is not possible

**if** (r > n):

**return** 0

    # Optimization for the cases when r is large

**if** (r > n **-** r):

        r **=** n **-** r

    # x stores the current result at

    x **=** 1

    # each iteration

    # Initialized to 1 as nC0 is always 1.

**for** i **in** range(1, r **+** 1):

        # Formula derived for calculating result is

        # C(n,r-1)\*(n-r+1)/r

        # Function calculates x\*(n-i+1) % p.

        x **=** moduloMultiplication(x, (n **+** 1 **-** i), p)

        # Function calculates x/i % p.

        x **=** modDivide(x, i, p)

**return** x

# Driver Code

n **=** 5

r **=** 3

p **=** 1000000007

**print**("Value of nCr % p is ", nCr(n, r, p))

# This code is contributed by phasing17

**Output**

Value of nCr % p is 10

**Complexity Analysis:**

* The above code needs an extra of O(1) space for the calculations.
* The time involved in the calculation of nCr % p is of the order O(n).

**19.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 follows, 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

Rod Cutting

**Method 1:**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}

#  A recursive solution for Rod cutting problem

#  Returns the best obtainable price for a rod of length n

#   and price[] as prices of different pieces

**def** cutRod(price, index, 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

    notCut **=** cutRod(price,index **-** 1,n)

    cut **=** float("-inf")

    rod\_length **=** index **+** 1

**if** (rod\_length <**=** n):

        cut **=** price[index]**+**cutRod(price,index,n **-** rod\_length)

**return** max(notCut, cut)

#  Driver program to test above functions

arr **=** [ 1, 5, 8, 9, 10, 17, 17, 20 ]

size **=** len(arr)

print("Maximum Obtainable Value is ",cutRod(arr, size **-** 1, size))

# This code is contributed by Vivek Maddeshiya

**Output**

Maximum Obtainable Value is 22

**Time Complexity:**O(2n) where n is the length of the price array.

**Space Complexity:**O(n) where n is the length of the price array.

**2) Overlapping Subproblems:**

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

The implementation simply follows the recursive structure mentioned above.

# A memoization solution for Rod cutting problem

 # Returns the best obtainable price for

  # a rod of length n and price[] as

  # prices of different pieces

**def** cutRoad(price,index,n,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

    notCut **=** cutRoad(price,index**-**1,n,dp)

    cut **= -**5

    rod\_length **=** index **+** 1

**if**(rod\_length <**=** n):

        cut **=** price[index] **+** cutRoad(price,index,n**-**rod\_length,dp)

    dp[index][n] **=** max(notCut,cut)

**return** dp[index][n]

# Driver program to test above functions

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1,5,8,9,10,17,17,20]

    size **=** len(arr)

    dp **=** []

    temp **=** []

**for** i **in** range(0,size**+**1):

        temp.append(**-**1)

**for** i **in** range(0,size):

        dp.append(temp)

    # print(dp)

**print**("Maximum Obtainable Value is :",end**=**' ')

**print**(cutRoad(arr,size**-**1,size,dp))

**Output**

Maximum Obtainable Value is 22

**Time Complexity:** O(n2)

**Auxiliary Space:** O(n2)+O(n)

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

cR() ---> cutRod()

cR(4)  
 / /   
 / /   
 cR(3) cR(2) cR(1) cR(0)  
 / | / |  
 / | / |   
 cR(2) cR(1) cR(0) cR(1) cR(0) cR(0)  
 / | |  
 / | |   
 cR(1) cR(0) cR(0) cR(0)  
 /  
 /  
CR(0)

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](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of the same subproblems can be avoided by constructing a temporary array val[] in a bottom-up manner.

# A Dynamic Programming solution for Rod cutting problem

INT\_MIN **= -**32767

# Returns the best obtainable price for a rod of length n and

# price[] as prices of different pieces

**def** cutRod(price, n):

    val **=** [0 **for** x **in** range(n**+**1)]

    val[0] **=** 0

    # Build the table val[] in bottom up manner and return

    # the last entry from the table

**for** i **in** range(1, n**+**1):

        max\_val **=** INT\_MIN

**for** j **in** range(i):

             max\_val **=** max(max\_val, price[j] **+** val[i**-**j**-**1])

        val[i] **=** max\_val

**return** val[n]

# Driver program to test above functions

arr **=** [1, 5, 8, 9, 10, 17, 17, 20]

size **=** len(arr)

**print**("Maximum Obtainable Value is " **+** str(cutRod(arr, size)))

# This code is contributed by Bhavya Jain

**Output**

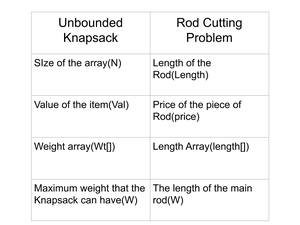
Maximum Obtainable Value is 22

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.

**Space Complexity:**O(n) as **val**array has been created.

**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.



# Python program for above approach

# Global Array for

# the purpose of memoization.

t **=** [[0 **for** i **in** range(9)] **for** j **in** range(9)]

# A recursive program, using ,

# memoization, to implement the

# rod cutting problem(Top-Down).

**def** un\_kp(price, length, Max\_len, n):

    # The maximum price will be zero,

    # when either the length of the rod

    # is zero or price is zero.

**if** (n **==** 0 **or** 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] **=** 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];

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    price **=** [1, 5, 8, 9, 10, 17, 17, 20 ];

    n **=**len(price);

    length **=** [0]**\***n;

**for** i **in** range(n):

        length[i] **=** i **+** 1;

    Max\_len **=** n;

    print("Maximum obtained value is " ,un\_kp(price, length, n, Max\_len));

# This code is contributed by gauravrajput1

**Output**

Maximum obtained value is 22

**Time Complexity:** O(n2)

**Auxiliary Space:** 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

#  Python program for above approach

**def** cutRod(prices, n):

    mat **=** [[0 **for** i **in** range(n**+**1)]**for** j **in** range(n**+**1)]

**for** i **in** range(1, n**+**1):

**for** j **in** range(1, n**+**1):

**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] **=** max(prices[i**-**1]**+**mat[i][j**-**i], mat[i**-**1][j])

**return** mat[n][n]

prices **=** [1, 5, 8, 9, 10, 17, 17, 20]

n **=** len(prices)

**print**("Maximum obtained value is ", cutRod(prices, n))

# This Code is Contributed By Vivek Maddeshiya

**Output**

Maximum obtained value is 22

**Time Complexity:** O(n2)

**Auxiliary Space:**O(n2)

**20.Painting Fence Algorithm**

Given a fence with n posts and k colors, find out the number of ways of painting the fence such that at most 2 adjacent posts have the same color. Since the answer can be large return it modulo 10^9 + 7.

**Examples:**

***Input :****n = 2 k = 4*

***Output :****16*

***Explanation:****We have 4 colors and 2 posts.*

*Ways when both posts have same color : 4*

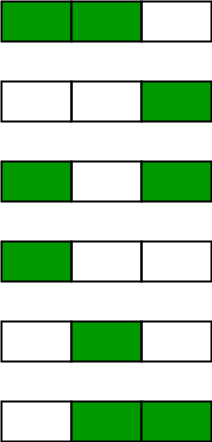
*Ways when both posts have diff color :4(choices for 1st post) \* 3(choices for 2nd post) = 12*

***Input :****n = 3 k = 2*

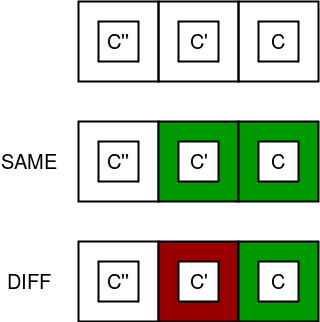
***Output :****6*

Painting the Fence

The following image depicts the 6 possible ways of painting 3 posts with 2 colors:



Consider the following image in which c, c’ and c” are the respective colors of posts i, i-1, and i -2.



According to the constraint of the problem, c = c’ = c” is not possible simultaneously, so either c’ != c or c” != c or both. There are k – 1 possibility for c’ != c and k – 1 for c” != c.

diff = no of ways when color of last  
 two posts is different  
 same = no of ways when color of last   
 two posts is same  
 total ways = diff + same

for n = 1  
 diff = k, same = 0  
 total = k

for n = 2  
 diff = k \* (k-1) //k choices for  
 first post, k-1 for next  
 same = k //k choices for common   
 color of two posts  
 total = k + k \* (k-1)

for n = 3  
 diff = k \* (k-1)\* (k-1)   
 //(k-1) choices for the first place   
 // k choices for the second place  
 //(k-1) choices for the third place  
 same = k \* (k-1) \* 2  
 // 2 is multiplied because consider two color R and B  
 // R R B or B R R   
 // B B R or R B B   
 c'' != c, (k-1) choices for it

Hence we deduce that,  
total[i] = same[i] + diff[i]  
same[i] = diff[i-1]  
diff[i] = (diff[i-1] + diff[i-2]) \* (k-1)  
 = total[i-1] \* (k-1)

Below is the implementation of the problem:

# Python3 program for Painting Fence Algorithm

# optimised version

# Returns count of ways to color k posts

**def** countWays(n, k):

    dp **=** [0] **\*** (n **+** 1)

    total **=** k

    mod **=** 1000000007

    dp[1] **=** k

    dp[2] **=** k **\*** k

**for** i **in** range(3,n**+**1):

        dp[i] **=** ((k **-** 1) **\*** (dp[i **-** 1] **+** dp[i **-** 2])) **%** mod

**return** dp[n]

# Driver code

n **=** 3

k **=** 2

print(countWays(n, k))

# This code is contributed by shubhamsingh10

**Output**

6

**Time Complexity:**O(N)

**Auxiliary Space:**O(N)

**Space optimization :**

We can optimize the above solution to use one variable instead of a table.

Below is the implementation of the problem:

# Python3 program for Painting

# Fence Algorithm

# Returns count of ways to color

# k posts using k colors

**def** countWays(n, k) :

    # There are k ways to color first post

    total **=** k

    mod **=** 1000000007

    # There are 0 ways for single post to

    # violate (same color\_ and k ways to

    # not violate (different color)

    same, diff **=** 0, k

    # Fill for 2 posts onwards

**for** i **in** range(2, n **+** 1) :

        # Current same is same as

        # previous diff

        same **=** diff

        # We always have k-1 choices

        # for next post

        diff **=** total **\*** (k **-** 1)

        diff **=** diff **%** mod

        # Total choices till i.

        total **=** (same **+** diff) **%** mod

**return** total

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_" :

    n, k **=** 3, 2

**print**(countWays(n, k))

# This code is contributed by Ryuga

**Output**

6

**Time Complexity:**O(N)

**Auxiliary Space:** O(1)

**21.Longest Common Subsequence (LCS)**

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

**Note:** A subsequence of a string is a sequence that is generated by deleting some characters (possibly 0) from the string without altering the order of the remaining characters. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, etc are subsequences of the string “abcdefg”.

**Examples:**

***Input:****S1 = “AGGTAB”, S2 = “GXTXAYB”*

***Output:****4*

***Explanation:****The longest subsequence which is present in both strings is “GTAB”.*

***Input:****S1 = “ABCDGH”, S2 = “AEDFHR”*

***Output:****3*

***Explanation:****The longest subsequence which is present in both strings is “ADH”.*

Recommended Problem

Longest Common Subsequence

**Naive Approach for LCS:**

The problem can be solved using recursion based on the following idea:

*Generate all the possible subsequences and find the longest among them that is present in both strings.*

**Time Complexity:** O(n \* 2n)

* Let us count the total subsequences with lengths 1, 2, . . . , n-1, n.
* From theory of permutation and combination we know number of combinations with 1 element is nC1. Number of combinations with 2 elements are nC2 and so on.
* So a string of length n has nC1 + nC2 + . . . nCn = 2n-1 different possible subsequences.
* Each subsequence takes O(n) time to compare.

**Auxiliary Space:** O(n) As we can reuse the same string.

**Recursive Approach for LCS:**

We can further improve the solution of LCS by utilizing the following observation:

*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 the LCS of the two strings X and Y.*

*Following is the recursive definition of L(X[0 . . . m-1], Y[0 . . . n-1]).*

* *If the 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 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]) )*

Follow the below steps to implement the idea:

* Create a recursive function [say **lcs()**].
* Check the relation between the last characters of the strings that are not yet processed.
* Depending on the relation call the next recursive function as mentioned above.
* Return the length of the LCS received as the answer.

Below is the implementation of the recursive approach:

# A Naive recursive Python implementation of LCS problem

**def** lcs(X, Y, m, n):

**if** m **==** 0 **or** n **==** 0:

**return** 0

**elif** 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))

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    S1 **=** "AGGTAB"

    S2 **=** "GXTXAYB"

    print("Length of LCS is", lcs(S1, S2, len(S1), len(S2)))

**Output**

Length of LCS is 4

**Time Complexity:** O(2n)

**Auxiliary Space:** O(1)

[Memoization](https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/)**Approach for LCS:**

If we notice carefully, we can observe that the above recursive solution holds the following two properties:

[Optimal Substructure](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)**:**

See for solving the structure of L(X[0, 1, . . ., m-1], Y[0, 1, . . . , n-1]) we are taking the help of the substructures of X[0, 1, …, m-2], Y[0, 1,…, n-2], depending on the situation (i.e., using them optimally) to find the solution of the whole.

[Overlapping Subproblems](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)**:**

If we use the above recursive approach for strings “**AXYT**” and “**AYZX**“, we will get a partial recursion tree as shown below. Here we can see that the subproblems L(“AXY”, “AYZ”) is being calculated more than once. If the total tree is considered there will be several such overlapping subproblems.

*L(“AXYT”, “AYZX”)*

*/                                     \*

*L(“AXY”, “AYZX”)                         L(“AXYT”, “AYZ”)*

*/                            \                             /                      \*

*L(“AX”, “AYZX”) L(“AXY”, “AYZ”)   L(“AXY”, “AYZ”)  L(“AXYT”, “AY”)*

**Approach:** Because of the presence of these two properties we can use Dynamic programming or Memoization to solve the problem. Below is the approach for solution using recursion.

*Create a recursive function. Also create a 2D array to store the result of a unique state. During the recursion call, if the same state is called more than once, then we can directly return the answer stored for that state instead of calculating again.*

Following is the memoization implementation for the LCS problem.

# A Top-Down DP implementation of LCS problem

# Returns length of LCS for X[0..m-1], Y[0..n-1]

**def** lcs(X, Y, m, n, dp):

**if** (m **==** 0 **or** n **==** 0):

**return** 0

**if** (dp[m][n] !**= -**1):

**return** dp[m][n]

**if** X[m **-** 1] **==** Y[n **-** 1]:

        dp[m][n] **=** 1 **+** lcs(X, Y, m **-** 1, n **-** 1, dp)

**return** dp[m][n]

    dp[m][n] **=** max(lcs(X, Y, m, n **-** 1, dp),lcs(X, Y, m **-** 1, n, dp))

**return** dp[m][n]

# Driver code

X **=** "AGGTAB"

Y **=** "GXTXAYB"

m **=** len(X)

n **=** len(Y)

dp **=** [[**-**1 **for** i **in** range(n **+** 1)]**for** j **in** range(m **+** 1)]

print(f"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(m \* n) where m and n are the string lengths.

**Auxiliary Space:**O(m \* n) here the recursive stack space is ignored.

[Dynamic Programming](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/)**for LCS:**

We can use the following steps to implement the dynamic programming approach for LCS.

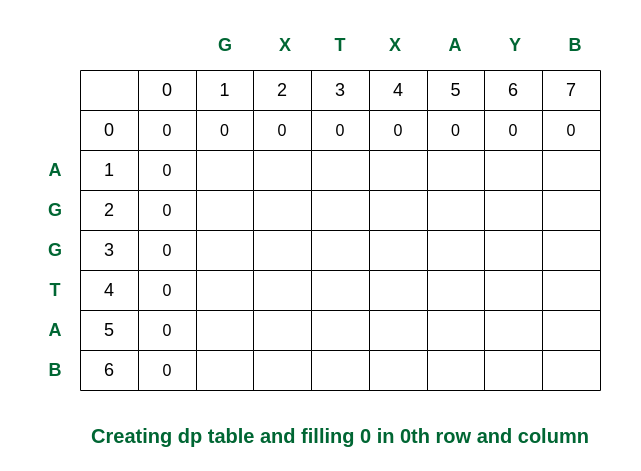
* Create a 2D array **dp[][]** with rows and columns equal to the length of each input string plus 1 [the number of rows indicates the indices of **S1** and the columns indicate the indices of **S2**].
* Initialize the first row and column of the dp array to 0.
* Iterate through the rows of the dp array, starting from 1 (say using iterator **i**).
* For each **i**, iterate all the columns from **j = 1 to n**:
* If **S1[i-1]** is equal to **S2[j-1]**, set the current element of the dp array to the value of the element to (**dp[i-1][j-1] + 1**).
* Else, set the current element of the dp array to the maximum value of **dp[i-1][j]** and **dp[i][j-1]**.
* After the nested loops, the last element of the dp array will contain the length of the LCS.

See the below illustration for a better understanding:

**Illustration:**

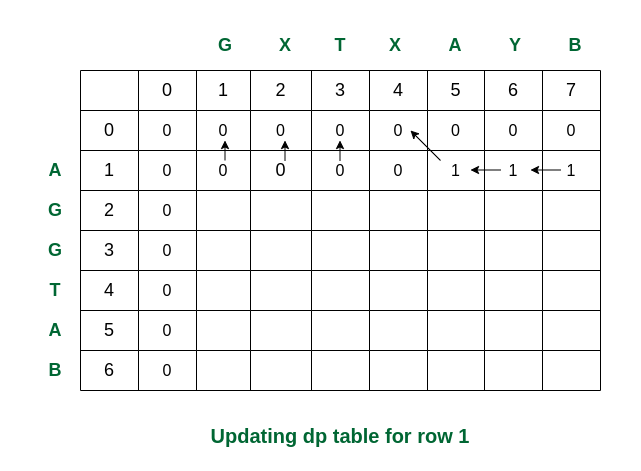
*Say the strings are****S1 = “AGGTAB”****and****S2 = “GXTXAYB”****.*

***First step:****Initially create a 2D matrix (say dp[][]) of  size 8 x 7 whose first row and first column are filled with 0.*



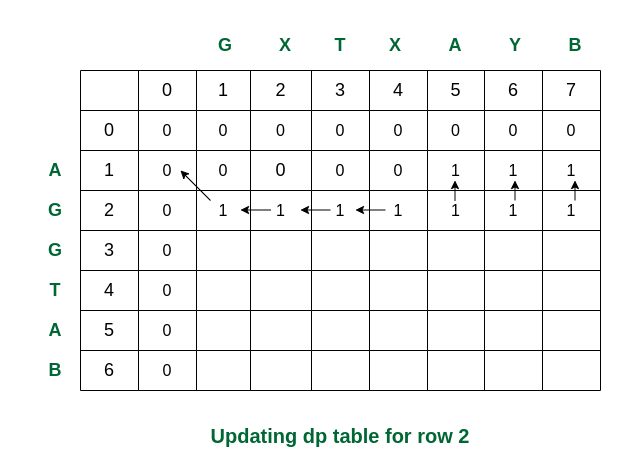
*Creating the dp table*

***Second step:****Traverse for i = 1. When j becomes 5, S1[0] and S2[4] are equal. So the dp[][] is updated. For the other elements take the maximum of dp[i-1][j] and dp[i][j-1]. (In this case, if both values are equal, we have used arrows to the previous rows).*



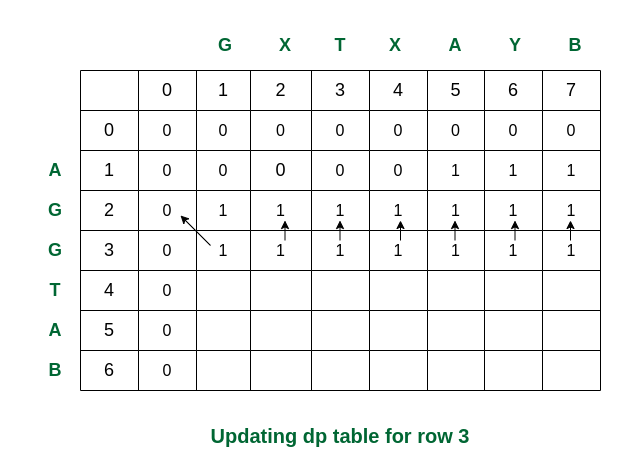
*Filling the row no 1*

***Third step:****While traversed for i = 2, S1[1] and S2[0] are the same (both are ‘G’). So the dp value in that cell is updated. Rest of the elements are updated as per the conditions.*



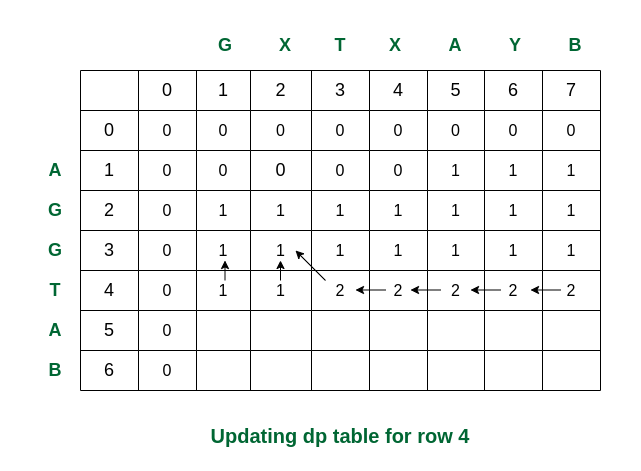
*Filling the row no. 2*

***Fourth step:****For i = 3, S1[2] and S2[0] are again same. The updations are as follows.*



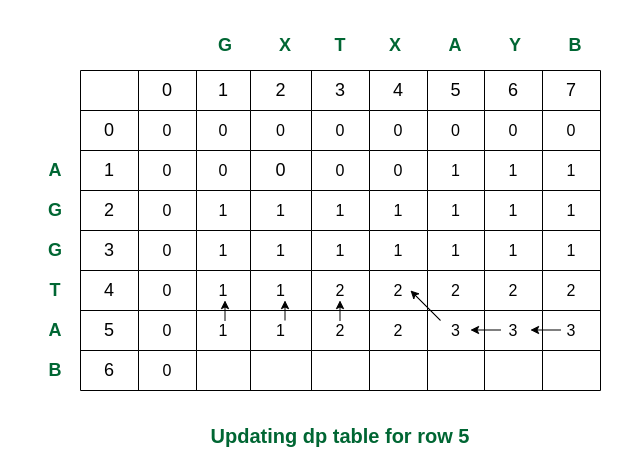
*Filling row no. 3*

***Fifth step:****For i = 4, we can see that S1[3] and S2[2] are same. So dp[4][3] updated as dp[3][2] + 1 = 2.*



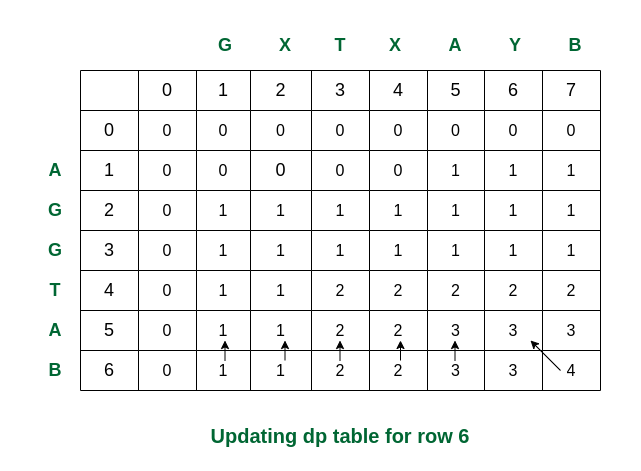
*Filling row 4*

***Sixth step:****Here we can see that for i = 5 and j = 5 the values of S1[4] and S2[4] are same (i.e., both are ‘A’). So dp[5][5] is updated accordingly and becomes 3.*



*Filling row 5*

***Final step:****For i = 6, see the last characters of both strings are same (they are ‘B’). Therefore the value of dp[6][7] becomes 4.*



*Filling the final row*

*So we get the maximum length of common subsequence as****4****.*

Following is a tabulated implementation for the LCS problem.

# Dynamic Programming implementation of LCS problem

**def** lcs(X , Y, m, n):

    # Declaring the array for storing the dp values

    L **=** [[None]**\***(n**+**1) **for** i **in** range(m**+**1)]

    # Following steps build L[m+1][n+1] in bottom up fashion

    # Note: L[i][j] contains length of LCS of X[0..i-1]

    # and Y[0..j-1]

**for** i **in** range(m**+**1):

**for** j **in** range(n**+**1):

**if** i **==** 0 **or** j **==** 0 :

                L[i][j] **=** 0

**elif** 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])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]

**return** L[m][n]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    S1 **=** "AGGTAB"

    S2 **=** "GXTXAYB"

    m **=** len(S1)

    n **=** len(S2)

**print** ("Length of LCS is", lcs(S1, S2, m, n) )

# This code is contributed by Nikhil Kumar Singh(nickzuck\_007)

**Output**

Length of LCS is 4

**Time Complexity:** O(m \* n) which is much better than the worst-case time complexity of Naive Recursive implementation.

**Auxiliary Space:** O(m \* n) because the algorithm uses an array of size (m+1)\*(n+1) to store the length of the common substrings.

**Usage of LCS Problem:**

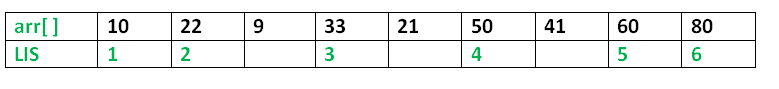
It is a classic computer science problem, the basis of [diff](http://en.wikipedia.org/wiki/Diff)(a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

**23.Longest Increasing Subsequence | DP-3**

We have already discussed [Overlapping Subproblems](https://www.geeksforgeeks.org/dynamic-programming-set-1/) and [Optimal Substructure](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/) 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}.



**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.](https://practice.geeksforgeeks.org/problems/longest-increasing-subsequence/0)

**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).

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:

# A naive Python implementation of LIS problem

""" 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 """

# global variable to store the maximum

**global** maximum

**def** \_lis(arr, n):

    # to allow the access of global variable

**global** maximum

    # Base Case

**if** n **==** 1:

**return** 1

    # maxEndingHere is the length of LIS ending with arr[n-1]

    maxEndingHere **=** 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** i **in** range(1, n):

        res **=** \_lis(arr, i)

**if** arr[i**-**1] < arr[n**-**1] **and** res**+**1 > maxEndingHere:

            maxEndingHere **=** res **+** 1

    # Compare maxEndingHere with overall maximum. And

    # update the overall maximum if needed

    maximum **=** max(maximum, maxEndingHere)

**return** maxEndingHere

**def** lis(arr):

    # to allow the access of global variable

**global** maximum

    # length of arr

    n **=** len(arr)

    # maximum variable holds the result

    maximum **=** 1

    # The function \_lis() stores its result in maximum

    \_lis(arr, n)

**return** maximum

# Driver program to test the above function

arr **=** [10, 22, 9, 33, 21, 50, 41, 60]

n **=** len(arr)

print ("Length of lis is ", lis(arr))

# This code is contributed by NIKHIL KUMAR SINGH

**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:

# Dynamic programming Python implementation

# of LIS problem

# lis returns length of the longest

# increasing subsequence in arr of size n

**def** lis(arr):

    n **=** len(arr)

    # Declare the list (array) for LIS and

    # initialize LIS values for all indexes

    lis **=** [1]**\***n

    # Compute optimized LIS values in bottom up manner

**for** i **in** range(1, n):

**for** j **in** range(0, i):

**if** arr[i] > arr[j] **and** lis[i] < lis[j] **+** 1:

                lis[i] **=** lis[j]**+**1

    # Initialize maximum to 0 to get

    # the maximum of all LIS

    maximum **=** 0

    # Pick maximum of all LIS values

**for** i **in** range(n):

        maximum **=** max(maximum, lis[i])

**return** maximum

# end of lis function

# Driver program to test above function

arr **=** [10, 22, 9, 33, 21, 50, 41, 60]

print ("Length of lis is", lis(arr))

# This code is contributed by Nikhil Kumar Singh

**Output**

Length of lis is 5

**Complexity Analysis:**

* **Time Complexity:** O(n2).   
  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)](https://www.geeksforgeeks.org/longest-monotonically-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}

# Dynamic Programming Approach of Finding LIS by reducing the problem to longest common Subsequence

**def** lis(a):

    n **=** len(a)

    # Creating the sorted list

    b **=** sorted(list(set(a)))

    m **=** len(b)

    # Creating dp table for storing the answers of sub problems

    dp **=** [[**-**1 **for** i **in** range(m**+**1)] **for** j **in** range(n**+**1)]

    # Finding Longest common Subsequence of the two arrays

**for** i **in** range(n**+**1):

**for** j **in** range(m**+**1):

**if** i **==** 0 **or** j **==** 0:

                dp[i][j] **=** 0

**elif** a[i**-**1] **==** b[j**-**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[**-**1][**-**1]

# Driver program to test above function

arr **=** [10, 22, 9, 33, 21, 50, 41, 60]

print("Length of lis is ", lis(arr))

# This code is Contributed by Dheeraj Khatri

**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

# A Naive Python recursive implementation

# of LIS problem

# 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

**import** sys

**def** f(idx, prev\_idx, n, a,dp):

**if** (idx **==** n):

**return** 0

**if** (dp[idx][prev\_idx **+** 1] !**= -**1):

**return** dp[idx][prev\_idx **+** 1]

    notTake **=** 0 **+** f(idx **+** 1, prev\_idx, n, a, dp)

    take **= -**sys.maxsize **-**1

**if** (prev\_idx **== -**1 **or** a[idx] > a[prev\_idx]):

        take **=** 1 **+** f(idx **+** 1, idx, n, a, dp)

    dp[idx][prev\_idx **+** 1] **=** max(take, notTake)

**return** dp[idx][prev\_idx **+** 1]

# Function to find length of longest increasing

# subsequence.

**def** longestSubsequence(n, a):

    dp **=** [[**-**1 **for** i **in** range(n **+** 1)]**for** j **in** range(n **+** 1)]

**return** f(0, **-**1, n, a, dp)

# Driver program to test above function

a **=** [ 3, 10, 2, 1, 20 ]

n **=** len(a)

print("Length of lis is ",longestSubsequence(n, a))

# This code is contributed by shinjanpatra

**Output**

Length of lis is 3

**Complexity Analysis:**

**Time Complexity:** O(n2).

**Auxiliary Space:** O(n2).

**24.Longest subsequence such that difference between adjacents is one**

Given an array of n size, the task is to find the longest subsequence such that difference between adjacents is one.

Examples:

**Input :**  arr[] = {10, 9, 4, 5, 4, 8, 6}  
**Output :**  3  
As longest subsequences with difference 1 are, "10, 9, 8",   
"4, 5, 4" and "4, 5, 6"

**Input :**  arr[] = {1, 2, 3, 2, 3, 7, 2, 1}  
**Output :**  7  
As longest consecutive sequence is "1, 2, 3, 2, 3, 2, 1"

Recommended Problem

Longest subsequence-1

[Dynamic Programming](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Dynamic%20Programming&sortBy=submissions)

[Algorithms](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Algorithms&sortBy=submissions)

[Flipkart](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Flipkart&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/longest-subsequence-such-that-difference-between-adjacents-is-one4724/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 37.6K

This problem is based upon the concept of [Longest Increasing Subsequence Problem](https://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/).

Let arr[0..n-1] be the input array and   
dp[i] be the length of the longest subsequence (with  
differences one) ending at index i such that arr[i]   
is the last element of the subsequence.

Then, dp[i] can be recursively written as:  
dp[i] = 1 + max(dp[j]) where 0 < j < i and   
 [arr[j] = arr[i] -1 or arr[j] = arr[i] + 1]  
dp[i] = 1, if no such j exists.

To find the result for a given array, we need   
to return max(dp[i]) where 0 < i < n.

Following is a Dynamic Programming based implementation. It follows the recursive structure discussed above.

# Function to find the length of longest subsequence

**def** longestSubseqWithDiffOne(arr, n):

    # Initialize the dp[] array with 1 as a

    # single element will be of 1 length

    dp **=** [1 **for** i **in** range(n)]

    # Start traversing the given array

**for** i **in** range(n):

        # Compare with all the previous elements

**for** j **in** range(i):

            # If the element is consecutive then

            # consider this subsequence and update

            # dp[i] if required.

**if** ((arr[i] **==** arr[j]**+**1) **or** (arr[i] **==** arr[j]**-**1)):

                dp[i] **=** max(dp[i], dp[j]**+**1)

    # Longest length will be the maximum value

    # of dp array.

    result **=** 1

**for** i **in** range(n):

**if** (result < dp[i]):

            result **=** dp[i]

**return** result

# Driver code

arr **=** [1, 2, 3, 4, 5, 3, 2]

# Longest subsequence with one difference is

# {1, 2, 3, 4, 3, 2}

n **=** len(arr)

**print** (longestSubseqWithDiffOne(arr, n))

# This code is contributed by Afzal Ansari

**Output**

6

**Time Complexity:**O(n2)

**Auxiliary Space:**O(n)

**Efficient Approach**

**def** longestSubsequence(A, N):

    L **=** [1]**\***N

    hm **=** {}

**for** i **in** range(1,N):

**if** abs(A[i]**-**A[i**-**1]) **==** 1:

            L[i] **=** 1 **+** L[i**-**1]

**elif** hm.get(A[i]**+**1,0) **or** hm.get(A[i]**-**1,0):

            L[i] **=** 1**+**max(hm.get(A[i]**+**1,0), hm.get(A[i]**-**1,0))

        hm[A[i]] **=** L[i]

**return** max(L)

# Driver code

A **=**  [1, 2, 3, 4, 5, 3, 2]

N **=** len(A)

print(longestSubsequence(A, N))

**Output**

6

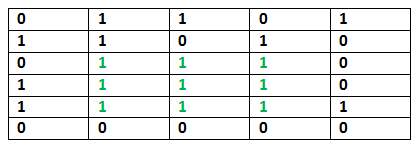
**Time Complexity :** O(n)

**Space Complexity :** O(n)

**25.Maximum size square sub-matrix with all 1s**

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.



Recommended Problem

Largest square formed in a matrix

Algorithm:

Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents the size of the square sub-matrix with all 1s including M[i][j] where M[i][j] is the rightmost and bottom-most entry in sub-matrix.

1) Construct a sum matrix S[R][C] for the given M[R][C].  
 a) Copy first row and first columns as it is from M[][] to S[][]  
 b) For other entries, use following expressions to construct S[][]  
 If M[i][j] is 1 then  
 S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1  
 Else /\*If M[i][j] is 0\*/  
 S[i][j] = 0  
2) Find the maximum entry in S[R][C]  
3) Using the value and coordinates of maximum entry in S[i], print   
 sub-matrix of M[][]

For the given M[R][C] in the above example, constructed S[R][C] would be:

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

The value of the maximum entry in the above matrix is 3 and the coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

# Python3 code for Maximum size

# square sub-matrix with all 1s

**def** printMaxSubSquare(M):

    R **=** len(M)  # no. of rows in M[][]

    C **=** len(M[0])  # no. of columns in M[][]

    S **=** []

**for** i **in** range(R):

        temp **=** []

**for** j **in** range(C):

**if** i **==** 0 **or** j **==** 0:

                temp **+=** M[i][j],

**else**:

                temp **+=** 0,

        S **+=** temp,

    # here we have set the first row and first column of S same as input matrix, other entries are set to 0

    # Update other entries

**for** i **in** range(1, R):

**for** j **in** range(1, C):

**if** (M[i][j] **==** 1):

                S[i][j] **=** min(S[i][j**-**1], S[i**-**1][j],

                              S[i**-**1][j**-**1]) **+** 1

**else**:

                S[i][j] **=** 0

    # Find the maximum entry and

    # indices of maximum entry in S[][]

    max\_of\_s **=** S[0][0]

    max\_i **=** 0

    max\_j **=** 0

**for** i **in** range(R):

**for** j **in** range(C):

**if** (max\_of\_s < S[i][j]):

                max\_of\_s **=** S[i][j]

                max\_i **=** i

                max\_j **=** j

    print("Maximum size sub-matrix is: ")

**for** i **in** range(max\_i, max\_i **-** max\_of\_s, **-**1):

**for** j **in** range(max\_j, max\_j **-** max\_of\_s, **-**1):

            print(M[i][j], end**=**" ")

        print("")

# Driver Program

M **=** [[0, 1, 1, 0, 1],

     [1, 1, 0, 1, 0],

     [0, 1, 1, 1, 0],

     [1, 1, 1, 1, 0],

     [1, 1, 1, 1, 1],

     [0, 0, 0, 0, 0]]

printMaxSubSquare(M)

# This code is contributed by Soumen Ghosh

**Output**

Maximum size sub-matrix is:   
1 1 1   
1 1 1   
1 1 1

**Time Complexity:** O(m\*n) where m is the number of rows and n is the number of columns in the given matrix.

**Auxiliary Space:** O(m\*n) where m is the number of rows and n is the number of columns in the given matrix.

**Algorithmic Paradigm:**Dynamic Programming

**Space Optimized Solution:**In order to compute an entry at any position in the matrix we only need the current row and the previous row.

# Python code for Maximum size square

# sub-matrix with all 1s

# (space optimized solution)

R **=** 6

C **=** 5

**def** printMaxSubSquare(M):

**global** R, C

    Max **=** 0

    # set all elements of S to 0 first

    S **=** [[0 **for** col **in** range(C)]**for** row **in** range(2)]

    # Construct the entries

**for** i **in** range(R):

**for** j **in** range(C):

            # Compute the entrie at the current position

            Entrie **=** M[i][j]

**if**(Entrie):

**if**(j):

                    Entrie **=** 1 **+** min(S[1][j **-** 1], min(S[0][j **-** 1], S[1][j]))

            # Save the last entrie and add the new one

            S[0][j] **=** S[1][j]

            S[1][j] **=** Entrie

            # Keep track of the max square length

            Max **=** max(Max, Entrie)

    # Print the square

**print**("Maximum size sub-matrix is: ")

**for** i **in** range(Max):

**for** j **in** range(Max):

**print**("1", end**=**" ")

**print**()

# Driver code

M **=** [[0, 1, 1, 0, 1],

     [1, 1, 0, 1, 0],

     [0, 1, 1, 1, 0],

     [1, 1, 1, 1, 0],

     [1, 1, 1, 1, 1],

     [0, 0, 0, 0, 0]]

printMaxSubSquare(M)

# This code is contributed by shinjanpatra

**Output**

Maximum size sub-matrix is:   
1 1 1   
1 1 1   
1 1 1

**Time Complexity**: O(m\*n) where m is the number of rows and n is the number of columns in the given matrix.

**Auxiliary space:** O(n) where n is the number of columns in the given matrix.

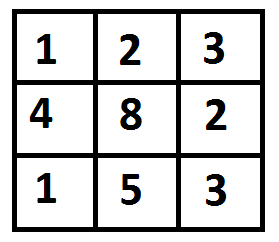
**26.Min Cost Path | DP-6**

Given a cost matrix cost[][] and a position (M, N) in cost[][], write a function that returns cost of minimum cost path to reach (M, N) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (M, N) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed.

**Note:**You may assume that all costs are positive integers.

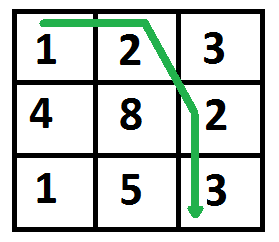
**Example:**

***Input:***



*The path with minimum cost is highlighted in the following figure. The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).*

***Output:***



[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/path-in-matrix/0)

**Min cost path using**[recursion](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)**:**

To solve the problem follow the below idea:

*This problem has the optimal substructure property. The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as “minimum of the 3 cells plus cost[m][n]”.*

*minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]*

Follow the below steps to solve the problem:

* If N is less than zero or M is less than zero then return Integer Maximum(Base Case)
* If M is equal to zero and N is equal to zero then return cost[M][N](Base Case)
* Return cost[M][N] + minimum of (minCost(M-1, N-1), minCost(M-1, N), minCost(M, N-1))

Below is the implementation of the above approach:

# A Naive recursive implementation of MCP(Minimum Cost Path) problem

**import** sys

R **=** 3

C **=** 3

# Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]

**def** minCost(cost, m, n):

**if** (n < 0 **or** m < 0):

**return** sys.maxsize

**elif** (m **==** 0 **and** n **==** 0):

**return** cost[m][n]

**else**:

**return** cost[m][n] **+** min(minCost(cost, m**-**1, n**-**1),

                                minCost(cost, m**-**1, n),

                                minCost(cost, m, n**-**1))

# A utility function that returns minimum of 3 integers \*/

**def** min(x, y, z):

**if** (x < y):

**return** x **if** (x < z) **else** z

**else**:

**return** y **if** (y < z) **else** z

# Driver code

cost **=** [[1, 2, 3],

        [4, 8, 2],

        [1, 5, 3]]

**print**(minCost(cost, 2, 2))

# This code is contributed by

# Smitha Dinesh Semwal

**Output**

8

**Time Complexity:**O((M \* N)3)

**Auxiliary Space:** O(M + N), for recursive stack space

**Min cost path using**[Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/)**:**

To solve the problem follow the below idea:

*It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which appear more than once. The time complexity of this naive recursive solution is exponential and it is terribly slow.*

*mC refers to minCost()  
 mC(2, 2)  
 / | \  
 / | \   
 mC(1, 1) mC(1, 2) mC(2, 1)  
 / | \ / | \ / | \   
 / | \ / | \ / | \  
 mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)*

*So the MCP problem has both properties (see*[*this*](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)*and*[*this*](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)*) of a dynamic programming problem. Like other typical*[*Dynamic Programming(DP) problems*](https://www.geeksforgeeks.org/archives/tag/dynamic-programming)*, recomputations of the same subproblems can be avoided by constructing a temporary array tc[][] in a bottom-up manner.*

Follow the below steps to solve the problem:

* Create a 2-D array ‘tc’ of size R \* C
* Calculate prefix sum for the first row and first column in ‘tc’ array as there is only one way to reach any cell in the first row or column
* Run a nested for loop for i [1, M] and j [1, N]
* Set tc[i][j] equal to minimum of (tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j]
* Return tc[M][N]

Below is the implementation of the above approach:

# Dynamic Programming Python implementation of Min Cost Path

# problem

R **=** 3

C **=** 3

**def** minCost(cost, m, n):

    # Instead of following line, we can use int tc[m+1][n+1] or

    # dynamically allocate memoery to save space. The following

    # line is used to keep te program simple and make it working

    # on all compilers.

    tc **=** [[0 **for** x **in** range(C)] **for** x **in** range(R)]

    tc[0][0] **=** cost[0][0]

    # Initialize first column of total cost(tc) array

**for** i **in** range(1, m**+**1):

        tc[i][0] **=** tc[i**-**1][0] **+** cost[i][0]

    # Initialize first row of tc array

**for** j **in** range(1, n**+**1):

        tc[0][j] **=** tc[0][j**-**1] **+** cost[0][j]

    # Construct rest of the tc array

**for** i **in** range(1, m**+**1):

**for** j **in** range(1, n**+**1):

            tc[i][j] **=** min(tc[i**-**1][j**-**1], tc[i**-**1][j], tc[i][j**-**1]) **+** cost[i][j]

**return** tc[m][n]

# Driver code

cost **=** [[1, 2, 3],

        [4, 8, 2],

        [1, 5, 3]]

print(minCost(cost, 2, 2))

# This code is contributed by Bhavya Jain

**Output**

8

**Time Complexity:** O(M \* N)

**Auxiliary Space:**O(M \* N)

**Min cost path using Dynamic Programming(Space optimized):**

To solve the problem follow the below idea:

*The idea is to use the same given/input array to store the solutions of subproblems in the above solution*

Follow the below steps to solve the problem:

* Calculate prefix sum for the first row and first column in ‘cost’ array as there is only one way to reach any cell in the first row or column
* Run a nested for loop for i [1, M-1] and j [1, N-1]
* Set cost[i][j] equal to minimum of (cost[i-1][j-1], cost[i-1][j], cost[i][j-1]) + cost[i][j]
* Return cost[M-1][N-1]

Below is the implementation of the above approach:

# Python3 program for the

# above approach

**def** minCost(cost, row, col):

    # For 1st column

**for** i **in** range(1, row):

        cost[i][0] **+=** cost[i **-** 1][0]

    # For 1st row

**for** j **in** range(1, col):

        cost[0][j] **+=** cost[0][j **-** 1]

    # For rest of the 2d matrix

**for** i **in** range(1, row):

**for** j **in** range(1, col):

            cost[i][j] **+=** (min(cost[i **-** 1][j **-** 1],

                               min(cost[i **-** 1][j],

                                   cost[i][j **-** 1])))

    # Returning the value in

    # last cell

**return** cost[row **-** 1][col **-** 1]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    row **=** 3

    col **=** 3

    cost **=** [[1, 2, 3],

            [4, 8, 2],

            [1, 5, 3]]

    print(minCost(cost, row, col))

# This code is contributed by Amit Katiyar

**Output**

8

**Time Complexity:**O(N \* M), where N is the number of rows and M is the number of columns

**Auxiliary Space:**O(1), since no extra space has been taken

**Min cost path using**[Dijkstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/)**:**

To solve the problem follow the below idea:

*We can also use the Dijkstra’s shortest path algorithm to find the path with minimum cost*

Follow the below steps to solve the problem:

* Create a 2-D dp array to store answer for each cell
* Declare a priority queue to perform dijkstra’s algorithm
* Return dp[M][N]

Below is the implementation of the approach:

# Minimum Cost Path using Dijkstra’s shortest path

#  algorithm with Min Heap by dinglizeng

# Python3

# Define the number of rows and the number of columns

R **=** 4

C **=** 5

# 8 possible moves

dx **=** [ 1, **-**1, 0, 0, 1, 1, **-**1, **-**1 ]

dy **=** [ 0, 0, 1, **-**1, 1, **-**1, 1, **-**1 ]

# The data structure to store the coordinates of

#  the unit square and the cost of path from the top

#  left.

**class** Cell():

**def** \_\_init\_\_(self, x, y, z):

        self.x **=** x

        self.y **=** y

        self.cost **=** z

# To verify whether a move is within the boundary.

**def** isSafe(x, y):

**return** (x >**=** 0 **and** x < R **and**

            y >**=** 0 **and** y < C)

# This solution is based on Dijkstra’s shortest

#  path algorithm

# For each unit square being visited, we examine all

#  possible next moves in 8 directions,

# calculate the accumulated cost of path for each

#  next move, adjust the cost of path of the adjacent

#  units to the minimum as needed.

# then add the valid next moves into a Min Heap.

# The Min Heap pops out the next move with the minimum

# accumulated cost of path.

# Once the iteration reaches the last unit at the lower

# right corner, the minimum cost path will be returned.

**def** minCost(cost, m, n):

    # the array to store the accumulated cost

    # of path from top left corner

    dp **=** [[0 **for** x **in** range(C)] **for** x **in** range(R)]

    # the array to record whether a unit

    # square has been visited

    visited **=** [[False **for** x **in** range(C)]

**for** x **in** range(R)]

    # Initialize these two arrays, set path cost

    # to maximum integer value, each unit as

    # not visited

**for** i **in** range(R):

**for** j **in** range(C):

            dp[i][j] **=** float("Inf")

            visited[i][j] **=** False

    # Define a reverse priority queue.

    # Priority queue is a heap based implementation.

    # The default behavior of a priority queue is

    # to have the maximum element at the top.

    # The compare class is used in the definition of

    # the Min Heap.

    pq **=** []

    # initialize the starting top left unit with the

    # cost and add it to the queue as the first move.

    dp[0][0] **=** cost[0][0]

    pq.append(Cell(0, 0, cost[0][0]))

**while**(len(pq)):

        # pop a move from the queue, ignore the units

        # already visited

        cell **=** pq[0]

        pq.pop(0)

        x **=** cell.x

        y **=** cell.y

**if**(visited[x][y]):

**continue**

        # mark the current unit as visited

        visited[x][y] **=** True

        # examine all non-visited adjacent units in 8

        # directions

        # calculate the accumulated cost of path for

        # each next move from this unit,

        # adjust the cost of path for each next

        # adjacent units to the minimum if possible.

**for** i **in** range(8):

            next\_x **=** x **+** dx[i]

            next\_y **=** y **+** dy[i]

**if**(isSafe(next\_x, next\_y) **and**

**not** visited[next\_x][next\_y]):

                dp[next\_x][next\_y] **=** min(dp[next\_x][next\_y],

                                        dp[x][y] **+** cost[next\_x][next\_y])

                pq.append(Cell(next\_x, next\_y,

                                dp[next\_x][next\_y]))

    # return the minimum cost path at the lower

    # right corner

**return** dp[m][n]

# Driver code

cost **=** [[1, 8, 8, 1, 5],

        [4, 1, 1, 8, 1],

        [4, 2, 8, 8, 1],

        [1, 5, 8, 8, 1]]

**print**(minCost(cost, 3, 4))

**Output**

7

**Time Complexity:**O(V + E \* logV), where V is (N\*M) and E is also (N\*M)

**Auxiliary Space:**O(N \* M)

**27.Minimum number of jumps to reach end**

Given an array **arr[]**where each element represents the max number of steps that can be made forward from that index. The task is to find the minimum number of jumps to reach the end of the array starting from index **0**. If the end isn’t reachable, return **-1**.

**Examples:**

***Input:****arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}*

***Output:****3 (1-> 3 -> 9 -> 9)*

***Explanation:****Jump from 1st element to 2nd element as there is only 1 step.*

*Now there are three options 5, 8 or 9. I*

*f 8 or 9 is chosen then the end node 9 can be reached. So 3 jumps are made.*

***Input:****arr[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}*

***Output:****10*

***Explanation:****In every step a jump is needed so the count of jumps is 10.*

Recommended Practice

[Jump Game](https://practice.geeksforgeeks.org/problems/jump-game/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/jump-game/1/)

**Minimum number of jumps to reach the end using**[Recursion](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)**:**

*Start from the first element and recursively call for all the elements reachable from the first element. The minimum number of jumps to reach end from first can be calculated using the minimum value from the recursive calls.*

***minJumps(start, end) = Min ( minJumps(k, end) )****for all k reachable from start.*

Follow the steps mentioned below to implement the idea:

* Create a recursive function.
* In each recursive call get all the reachable nodes from that index.
* For each of the index call the recursive function.
* Find the minimum number of jumps to reach the end from current index.
* Return the minimum number of jumps from the recursive call.

Below is the Implementation of the above approach:

# Python3 program to find Minimum

# number of jumps to reach end

# Returns minimum number of jumps

# to reach arr[h] from arr[l]

**def** minJumps(arr, l, h):

    # Base case: when source and

    # destination are same

**if** (h **==** l):

**return** 0

    # when nothing is reachable

    # from the given source

**if** (arr[l] **==** 0):

**return** float('inf')

    # Traverse through all the points

    # reachable from arr[l]. Recursively

    # get the minimum number of jumps

    # needed to reach arr[h] from

    # these reachable points.

    min **=** float('inf')

**for** i **in** range(l **+** 1, h **+** 1):

**if** (i < l **+** arr[l] **+** 1):

            jumps **=** minJumps(arr, i, h)

**if** (jumps !**=** float('inf') **and**

                    jumps **+** 1 < min):

                min **=** jumps **+** 1

**return** min

# Driver program to test above function

arr **=** [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]

n **=** len(arr)

print('Minimum number of jumps to reach',

      'end is', minJumps(arr, 0, n**-**1))

# This code is contributed by Soumen Ghosh

**Output**

Minimum number of jumps to reach the end is 3

**Time complexity:** O(nNn).

* There are maximum n possible ways to move from an element.
* So the maximum number of steps can be nn, Thus O(nn)

**Auxiliary Space:** O(n). For recursion call stack.

**Minimum number of jumps to reach end using**[Dynamic Programming](https://www.geeksforgeeks.org/dynamic-programming/)**from left to right:**

*It can be observed that there will be overlapping subproblems.*

*For example in array, arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9} minJumps(3, 9) will be called two times as arr[3] is reachable from arr[1] and arr[2]. So this problem has both properties (*[*optimal substructure*](https://www.geeksforgeeks.org/archives/12819)*and*[*overlapping subproblems*](https://www.geeksforgeeks.org/archives/12635)*) of Dynamic Programming*

Follow the below steps to implement the idea:

* Create jumps[] array from **left to right**such that jumps[i] indicate the minimum number of jumps needed to reach arr[i] from arr[0].
* To fill the jumps array run a nested loop inner loop counter is **j**and the outer loop count is **i**.
* Outer loop from **1 to n-1** and inner loop from **0 to i**.
* If i is less than j + arr[j] then set jumps[i] to minimum of jumps[i] and jumps[j] + 1. initially set jump[i] to INT MAX
* Return jumps[n-1].

Below is the implementation of the above approach:

# Python3 program to find Minimum

# number of jumps to reach end

# Returns minimum number of jumps

# to reach arr[n-1] from arr[0]

**def** minJumps(arr, n):

    jumps **=** [0 **for** i **in** range(n)]

**if** (n **==** 0) **or** (arr[0] **==** 0):

**return** float('inf')

    jumps[0] **=** 0

    # Find the minimum number of

    # jumps to reach arr[i] from

    # arr[0] and assign this

    # value to jumps[i]

**for** i **in** range(1, n):

        jumps[i] **=** float('inf')

**for** j **in** range(i):

**if** (i <**=** j **+** arr[j]) **and** (jumps[j] !**=** float('inf')):

                jumps[i] **=** min(jumps[i], jumps[j] **+** 1)

**break**

**return** jumps[n**-**1]

# Driver Program to test above function

arr **=** [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]

size **=** len(arr)

print('Minimum number of jumps to reach',

      'end is', minJumps(arr, size))

# This code is contributed by Soumen Ghosh

**Output**

Minimum number of jumps to reach end is 3

Thanks to paras for suggesting this method.

**Time Complexity:** O(n2)

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

**Another implementation using Dynamic programming:**

*Build****jumps[]****array from****right to left****such that****jumps[i]****indicate the minimum number of jumps needed to reach****arr[n-1]****from arr[i]. Finally, we return jumps[0]. Use Dynamic programming in a similar way of the above method.*

Below is the Implementation of the above approach:

# Python3 program to find Minimum

# number of jumps to reach end

# Returns Minimum number of

# jumps to reach end

**def** minJumps(arr, n):

    # jumps[0] will hold the result

    jumps **=** [0 **for** i **in** range(n)]

    # Minimum number of jumps needed

    # to reach last element from

    # last elements itself is always 0

    # jumps[n-1] is also initialized to 0

    # Start from the second element,

    # move from right to left and

    # construct the jumps[] array where

    # jumps[i] represents minimum number

    # of jumps needed to reach arr[m-1]

    # form arr[i]

**for** i **in** range(n**-**2, **-**1, **-**1):

        # If arr[i] is 0 then arr[n-1]

        # can't be reached from here

**if** (arr[i] **==** 0):

            jumps[i] **=** float('inf')

        # If we can directly reach to

        # the end point from here then

        # jumps[i] is 1

**elif** (arr[i] >**=** n **-** i **-** 1):

            jumps[i] **=** 1

        # Otherwise, to find out the

        # minimum number of jumps

        # needed to reach arr[n-1],

        # check all the points

        # reachable from here and

        # jumps[] value for those points

**else**:

            # initialize min value

            min **=** float('inf')

            # following loop checks with

            # all reachable points and

            # takes the minimum

**for** j **in** range(i **+** 1, n):

**if** (j <**=** arr[i] **+** i):

**if** (min > jumps[j]):

                        min **=** jumps[j]

            # Handle overflow

**if** (min !**=** float('inf')):

                jumps[i] **=** min **+** 1

**else**:

                # or INT\_MAX

                jumps[i] **=** min

**return** jumps[0]

# Driver program to test above function

arr **=** [1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9]

n **=** len(arr)

print('Minimum number of jumps to reach',

      'end is', minJumps(arr, n**-**1))

# This code is contributed by Soumen Ghosh

**Output**

Minimum number of jumps to reach end is 3

**Time complexity:**O(n2). Nested traversal of the array is needed.

**Auxiliary Space:**O(n). To store the DP array linear space is needed.

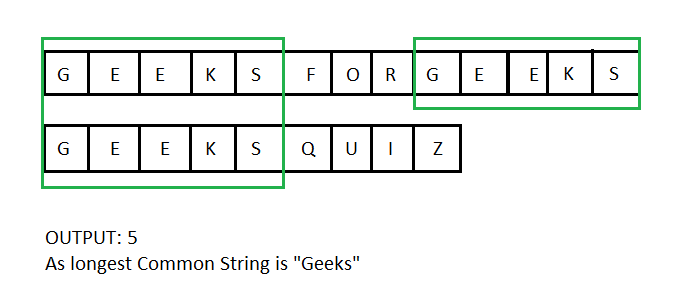
**28.Longest Common Substring (Space optimized DP solution)**

Given two strings ‘X’ and ‘Y’, find the length of longest common substring. Expected space complexity is linear.

**Examples :**

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

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



[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We have discussed [Dynamic programming based solution](https://www.geeksforgeeks.org/longest-common-substring/) for Longest common substring. The auxiliary space used by the solution is O(m\*n), where m and n are lengths of string X and Y. The space used by solution can be reduced to O(2\*n).

Suppose we are at position mat[i][j]. Now if X[i-1] == Y[j-1], then we add the value of mat[i-1][j-1] to our result. That is we add value from previous row and value for all other rows below the previous row are never used. So, at a time we are using only two consecutive rows. This observation can be used to reduce the space required to find length of longest common substring.

Instead of creating a matrix of size m\*n, we create a matrix of size 2\*n. A variable currRow is used to represent that either row 0 or row 1 of this matrix is currently used to find length. Initially row 0 is used as current row for the case when length of string X is zero. At the end of each iteration, current row is made previous row and previous row is made new current row.

# Space optimized Python3 implementation

# of longest common substring.

**import** numpy as np

# Function to find longest common substring.

**def** LCSubStr(X, Y) :

    # Find length of both the strings.

    m **=** len(X)

    n **=** len(Y)

    # Variable to store length of

    # longest common substring.

    result **=** 0

    # Matrix to store result of two

    # consecutive rows at a time.

    len\_mat **=** np.zeros((2, n))

    # Variable to represent which row

    # of matrix is current row.

    currRow **=** 0

    # For a particular value of i and j,

    # len\_mat[currRow][j] stores length of

    # longest common substring in string

    # X[0..i] and Y[0..j].

**for** i **in** range(m) :

**for** j **in** range(n) :

**if** (i **==** 0 | j **==** 0) :

                len\_mat[currRow][j] **=** 0

**elif** (X[i **-** 1] **==** Y[j **-** 1]) :

                len\_mat[currRow][j] **=** len\_mat[1 **-** currRow][j **-** 1] **+** 1

                result **=** max(result, len\_mat[currRow][j])

**else** :

                len\_mat[currRow][j] **=** 0

        # Make current row as previous row and

        # previous row as new current row.

        currRow **=** 1 **-** currRow

**return** result

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_" :

    X **=** "GeeksforGeeks"

    Y **=** "GeeksQuiz"

    print(LCSubStr(X, Y))

# This code is contributed by Ryuga

**Output :**

5

**Time Complexity:**O(m\*n)

**Auxiliary Space:**O(n)

**29.Count ways to reach the nth stair using step 1, 2 or 3**

A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

**Examples:**

**Input :** 4  
**Output :** 7  
**Explanation:**  
Below are the seven ways  
 1 step + 1 step + 1 step + 1 step  
 1 step + 2 step + 1 step  
 2 step + 1 step + 1 step   
 1 step + 1 step + 2 step  
 2 step + 2 step  
 3 step + 1 step  
 1 step + 3 step

**Input :** 3  
**Output :** 4  
**Explanation:**  
Below are the four ways  
 1 step + 1 step + 1 step  
 1 step + 2 step  
 2 step + 1 step  
 3 step

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**There are two methods to solve this problem:**

1. Recursive Method
2. Dynamic Programming

**Method 1:** Recursive.

There are n stairs, and a person is allowed to jump next stair, skip one stair or skip two stairs. So there are n stairs. So if a person is standing at i-th stair, the person can move to i+1, i+2, i+3-th stair. A recursive function can be formed where at current index i the function is recursively called for i+1, i+2 and i+3 th stair.

There is another way of forming the recursive function. To reach a stair i, a person has to jump either from i-1, i-2 or i-3 th stair or i is the starting stair.

**Algorithm:**

1. Create a recursive function (count(int n)) which takes only one parameter.
2. Check the base cases. If the value of n is less than 0 then return 0, and if the value of n is equal to zero then return 1 as it is the starting stair.
3. Call the function recursively with values n-1, n-2 and n-3 and sum up the values that are returned, i.e. sum = count(n-1) + count(n-2) + count(n-3)
4. Return the value of the sum.

# Python program to find n-th stair

# using step size 1 or 2 or 3.

# Returns count of ways to reach n-th

# stair using 1 or 2 or 3 steps.

**def** findStep(n):

**if** ( n **==** 0 ):

**return** 1

**elif** (n < 0):

**return** 0

**else**:

**return** findStep(n **-** 3) **+** findStep(n **-** 2) **+** findStep(n **-** 1)

# Driver code

n **=** 4

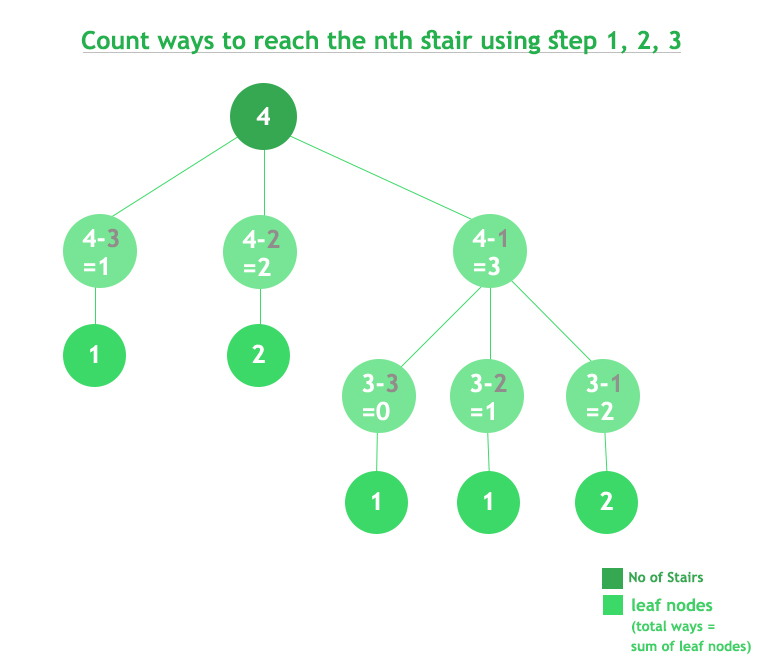
print(findStep(n))

# This code is contributed by Nikita Tiwari.

**Output**

7

**Working:**



**Complexity Analysis:**

* **Time Complexity:** O(3n).   
  The time complexity of the above solution is exponential, a close upper bound will be O(3n). From each state, 3 recursive function are called. So the upperbound for n states is O(3n).
* **Space Complexity:**O(1).   
  As no extra space is required.

**Note:** The Time Complexity of the program can be optimized using Dynamic Programming.

**Method 2:** Dynamic Programming.

The idea is similar, but it can be observed that there are n states but the recursive function is called 3 ^ n times. That means that some states are called repeatedly. So the idea is to store the value of states. This can be done in two ways.

* *Top-Down Approach:* The first way is to keep the recursive structure intact and just store the value in a HashMap and whenever the function is called again return the value store without computing ().
* *Bottom-Up Approach:* The second way is to take an extra space of size n and start computing values of states from 1, 2 .. to n, i.e. compute values of i, i+1, i+2 and then use them to calculate the value of i+3.

**Algorithm:**

1. Create an array of size n + 1 and initialize the first 3 variables with 1, 1, 2. The base cases.
2. Run a loop from 3 to n.
3. For each index i, computer value of ith position as dp[i] = dp[i-1] + dp[i-2] + dp[i-3].
4. Print the value of dp[n], as the Count of the number of ways to reach n th step.

# Python program to find n-th stair

# using step size 1 or 2 or 3.

# A recursive function used by countWays

**def** countWays(n):

    res **=** [0] **\*** (n **+** 2)

    res[0] **=** 1

    res[1] **=** 1

    res[2] **=** 2

**for** i **in** range(3, n **+** 1):

        res[i] **=** res[i **-** 1] **+** res[i **-** 2] **+** res[i **-** 3]

**return** res[n]

# Driver code

n **=** 4

print(countWays(n))

# This code is contributed by Nikita Tiwari.

**Output**

7

* **Working:**

1 -> 1 -> 1 -> 1  
1 -> 1 -> 2  
1 -> 2 -> 1  
1 -> 3  
2 -> 1 -> 1  
2 -> 2  
3 -> 1

So Total ways: 7

* **Complexity Analysis:**
* **Time Complexity:**O(n).   
  Only one traversal of the array is needed. So Time Complexity is O(n).
* **Space Complexity:**O(n).   
  To store the values in a DP, n extra space is needed.

**Method 3: Matrix Exponentiation (O(logn) Approach)**

Matrix Exponentiation is mathematical ways to solve DP problem in better time complexity. Matrix Exponentiation Technique has Transformation matrix of Size K X K and Functional Vector (K X 1) .By taking n-1th power of Transformation matrix and Multiplying It With functional vector Give Resultant Vector say it Res of Size K X 1. First Element of Res will be Answer for given n value. This Approach Will Take O(K^3logn) Time Complexity Which Is Complexity of Finding (n-1) power of Transformation Matrix.

**Key Terms:**

K = No of Terms in which F(n) depend ,from Recurrence Relation We can Say That F(n) depend On F(n-1) and F(n-2). => K =3

F1 =  Vector (1D array) that contain F(n) value of First K terms. Since K=3 =>F1 will have F(n) value of first 2 terms. F1=[1,2,4]

T = Transformation Matrix that is a 2D matrix of Size K X K and  Consist Of All 1 After Diagonal And Rest All Zero except last row. Last Row Will have coefficient Of all K terms in which F(n)  depends In Reverse Order. => T =[ [0 1 0] ,[0 0 1], [1 1 1] ].

**Algorithms:**

1)Take Input N  
2)If N < K then Return Precalculated Answer //Base Condition  
3)construct F1 Vector and T (Transformation Matrix)  
4)Take N-1th power of T by using Optimal Power(T,N) Methods and assign it in T  
5)return (TXF)[1]

for Optimal Power(T, N) Methods Refer Following

Article: <https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/>

k **=** 3

# Multiply Two Matrix Function

**def** multiply(A, B):

    # third matrix to store multiplication of Two matrix9\*

    C **=** [[0 **for** x **in** range(k**+**1)] **for** y **in** range(k**+**1)]

**for** i **in** range(1, k**+**1):

**for** j **in** range(1, k**+**1):

**for** x **in** range(1, k**+**1):

                C[i][j] **=** (C[i][j] **+** (A[i][x] **\*** B[x][j]))

**return** C

# Optimal Way For finding pow(t,n)

# If n Is Odd then It Will be t\*pow(t,n-1)

# else return pow(t,n/2)\*pow(t,n/2)

**def** pow(t,  n):

    # base Case

**if** (n **==** 1):

**return** t

    # Recurrence Case

**if** (n & 1):

**return** multiply(t, pow(t, n **-** 1))

**else**:

        X **=** pow(t, n **//** 2)

**return** multiply(X, X)

**def** compute(n):

    # base Case

**if** (n **==** 0):

**return** 1

**if** (n **==** 1):

**return** 1

**if** (n **==** 2):

**return** 2

    # Function Vector(indexing 1 )

    # that is [1,2]

    f1 **=** [0]**\***(k **+** 1)

    f1[1] **=** 1

    f1[2] **=** 2

    f1[3] **=** 4

    # Constructing Transformation Matrix that will be

    # [[0,1,0],[0,0,1],[3,2,1]]

    t **=** [[0 **for** x **in** range(k**+**1)] **for** y **in** range(k**+**1)]

**for** i **in** range(1, k**+**1):

**for** j **in** range(1, k**+**1):

**if** (i < k):

                # Store 1 in cell that is next to diagonal of Matrix else Store 0 in

                # cell

**if** (j **==** i **+** 1):

                    t[i][j] **=** 1

**else**:

                    t[i][j] **=** 0

**continue**

            # Last Row - store the Coefficients in reverse order

            t[i][j] **=** 1

    # Computing T^(n-1) and Setting Transformation matrix T to T^(n-1)

    t **=** pow(t, n **-** 1)

    sum **=** 0

    # Computing first cell (row=1,col=1) For Resultant Matrix TXF

**for** i **in** range(1, k**+**1):

        sum **+=** t[1][i] **\*** f1[i]

**return** sum

# Driver Code

n **=** 4

print(compute(n))

n **=** 5

print(compute(n))

n **=** 10

print(compute(n))

# This code is contributed by Shubhamsingh10

**Output**

7  
13  
274

Explanation:  
We Know For This Question   
Transformation Matrix M= [[0,1,0],[0,0,1],[1,1,1]]  
Functional Vector F1 = [1,2,4]  
for n=2 :  
 ans = (M X F1)[1]   
 ans = [2,4,7][1]   
 ans = 2 //[2,4,7][1] = First cell value of [2,4,7] i.e 2  
for n=3 :  
 ans = (M X M X F1)[1] //M^(3-1) X F1 = M X M X F1  
 ans = (M X [2,4,7])[1]   
 ans = [4,7,13][1]  
 ans = 4  
for n = 4 :  
 ans = (M^(4-1) X F1)[1]  
 ans = (M X M X M X F1) [1]   
 ans = (M X [4,7,13])[1]   
 ans = [7,13,24][1]  
 ans = 7  
for n = 5 :  
 ans = (M^4 X F1)[1]  
 ans = (M X [7,13,24])[1]  
 ans = [13,24,44][1]  
 ans = 13

**Time Complexity:**

O(K^3log(n)) //For Computing pow(t,n-1)  
For this question K is 3  
So Overall Time Complexity is O(27log(n))=O(logn)

**Auxiliary Space**: O(n^2) because extra space of vector have been used

**Method 4: Using four variables**

The idea is based on the Fibonacci series but here with 3 sums. we will hold the values of the first three stairs in 3 variables and will use the fourth variable to find the number of ways.

# A Python program to count number of ways

# to reach nth stair when

# A recursive function used by countWays

**def** countWays(n):

    # declaring three variables

    # and holding the ways

    # for first three stairs

    a **=** 1

    b **=** 2

    c **=** 4

    d **=** 0 # fourth variable

**if** (n **==** 0 **or** n **==** 1 **or** n **==** 2):

**return** n

**if** (n **==** 3):

**return** c

**for** i **in** range(4,n**+**1):

        # starting from 4 as

        d **=** c **+** b **+** a # already counted for 3 stairs

        a **=** b

        b **=** c

        c **=** d

**return** d

# Driver program to test above functions

n **=** 4

print(countWays(n))

# This code is contributed by shivanisinghss2110

**Output**

7

**Time Complexity:**O(n)

**Auxiliary** **Space:** O(1), since no extra space has been taken.

**Method 5:** DP using memoization(Top down approach)

We can avoid the repeated work done in method 1(recursion) by storing the number of ways calculated so far.

We just need to store all the values in an array.

# Python Program to find n-th stair using step size

# 1 or 2 or 3.

**class** GFG:

**def** findStepHelper(self, n, dp):

        # Base Case

**if** (n **==** 0):

**return** 1

**elif** (n < 0):

**return** 0

        # If subproblems are already calculated

        #then return it

**if** (dp[n] !**= -**1):

**return** dp[n]

        # store the subproblems in the vector

        dp[n] **=** self.findStepHelper(n **-** 3, dp) **+** self.findStepHelper(n **-** 2, dp) **+** self.findStepHelper(n **-** 1, dp)

**return** dp[n]

    # Returns count of ways to reach n-th stair

    # using 1 or 2 or 3 steps.

**def** findStep(self, n):

        dp **=** [**-**1 **for** i **in** range(n **+** 1)]

**return** self.findStepHelper(n, dp)

# Driver code

g **=** GFG()

n **=** 4

**print**(g.findStep(n))

# This code is contributed by shinjanpatra.

**Output**

7

**Complexity Analysis:**

* **Time Complexity:** O(n). Only one traversal of the array is needed. So Time Complexity is O(n).
* **Space Complexity:** O(n). To store the values in a DP, n extra space is needed. Also, stack space for recursion is needed which is again O(n)

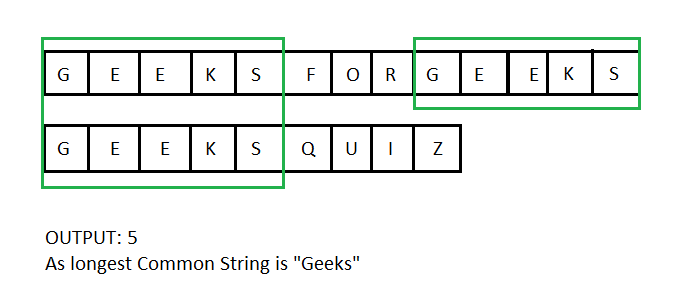
**30.Longest Common Substring (Space optimized DP solution)**

Given two strings ‘X’ and ‘Y’, find the length of longest common substring. Expected space complexity is linear.

**Examples :**

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

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



[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We have discussed [Dynamic programming based solution](https://www.geeksforgeeks.org/longest-common-substring/) for Longest common substring. The auxiliary space used by the solution is O(m\*n), where m and n are lengths of string X and Y. The space used by solution can be reduced to O(2\*n).

Suppose we are at position mat[i][j]. Now if X[i-1] == Y[j-1], then we add the value of mat[i-1][j-1] to our result. That is we add value from previous row and value for all other rows below the previous row are never used. So, at a time we are using only two consecutive rows. This observation can be used to reduce the space required to find length of longest common substring.

Instead of creating a matrix of size m\*n, we create a matrix of size 2\*n. A variable currRow is used to represent that either row 0 or row 1 of this matrix is currently used to find length. Initially row 0 is used as current row for the case when length of string X is zero. At the end of each iteration, current row is made previous row and previous row is made new current row.

# Space optimized Python3 implementation

# of longest common substring.

**import** numpy as np

# Function to find longest common substring.

**def** LCSubStr(X, Y) :

    # Find length of both the strings.

    m **=** len(X)

    n **=** len(Y)

    # Variable to store length of

    # longest common substring.

    result **=** 0

    # Matrix to store result of two

    # consecutive rows at a time.

    len\_mat **=** np.zeros((2, n))

    # Variable to represent which row

    # of matrix is current row.

    currRow **=** 0

    # For a particular value of i and j,

    # len\_mat[currRow][j] stores length of

    # longest common substring in string

    # X[0..i] and Y[0..j].

**for** i **in** range(m) :

**for** j **in** range(n) :

**if** (i **==** 0 | j **==** 0) :

                len\_mat[currRow][j] **=** 0

**elif** (X[i **-** 1] **==** Y[j **-** 1]) :

                len\_mat[currRow][j] **=** len\_mat[1 **-** currRow][j **-** 1] **+** 1

                result **=** max(result, len\_mat[currRow][j])

**else** :

                len\_mat[currRow][j] **=** 0

        # Make current row as previous row and

        # previous row as new current row.

        currRow **=** 1 **-** currRow

**return** result

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_" :

    X **=** "GeeksforGeeks"

    Y **=** "GeeksQuiz"

    print(LCSubStr(X, Y))

# This code is contributed by Ryuga

**Output :**

5

**Time Complexity:**O(m\*n)

**Auxiliary Space:**O(n)

**31.Count all possible paths from top left to bottom right of a mXn matrix**

The problem is to count all the possible paths from the top left to the bottom right of a M X N matrix with the constraints that from each cell you can either move only to the right or down

**Examples:**

***Input:****M = 2, N = 2*

***Output:****2*

***Explanation:****There are two paths*

*(0, 0) -> (0, 1) -> (1, 1)*

*(0, 0) -> (1, 0) -> (1, 1)*

***Input:****M = 2, N = 3*

***Output:****3*

***Explanation:****There are three paths*

*(0, 0) -> (0, 1) -> (0, 2) -> (1, 2)*

*(0, 0) -> (0, 1) -> (1, 1) -> (1, 2)*

*(0, 0) -> (1, 0) -> (1, 1) -> (1, 2)*

Recommended Problem

Number of paths

**32.Count all possible paths from top left to the bottom right of a M X N matrix using**[Recursion](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)**:**

To solve the problem follow the below idea:

*We can recursively move to right and down from the start until we reach the destination and then add up all valid paths to get the answer.*

Follow the below steps to solve the problem:

* Create a recursive function with parameters as row and column index
* Call this recursive function for N-1 and M-1
* In the recursive function
* If N == 1 or M == 1 then return 1
* else call the recursive function with (N-1, M) and (N, M-1) and return the sum of this
* Print the answer

Below is the implementation of the above approach:

# Python program to count all possible paths

# from top left to bottom right

# function to return count of possible paths

# to reach cell at row number m and column

# number n from the topmost leftmost

# cell (cell at 1, 1)

**def** numberOfPaths(m, n):

    # If either given row number is first

    # or given column number is first

**if**(m **==** 1 **or** n **==** 1):

**return** 1

# If diagonal movements are allowed

# then the last addition

# is required.

**return** numberOfPaths(m**-**1, n) **+** numberOfPaths(m, n**-**1)

# Driver program to test above function

**if** \_\_name\_\_ **==** '\_\_main\_\_':

  m **=** 3

  n **=** 3

  print(numberOfPaths(m, n))

# This code is contributed by Aditi Sharma

**Output**

6

**Time Complexity:**O(2N)

**Auxiliary Space:**O(N + M)

**33.Count all possible paths from top left to the bottom right of a M X N matrix using**[Memoization](https://www.geeksforgeeks.org/what-is-memoization-a-complete-tutorial/)**:**

To solve the problem follow the below idea:

*As the above recursive solution has overlapping subproblems so we can declare a 2-D array to save the values for different states of the recursive function and later on use the values of this dp array to get the answer for already solved subproblems*

Follow the below steps to solve the problem:

* Declare a 2-D array of size N X M
* Create a recursive function with parameters as row and column index and 2-D array
* Call this recursive function for N-1 and M-1
* In the recursive function
* If N == 1 or M == 1 then return 1
* If the value of this recursive function is not stored in the 2-D array then call the recursive function for (N-1, M, dp) and (N, M-1, dp) and assign the sum of answers of these functions in the 2-D array and return this value
* else return the value of this function stored in the 2-D array
* Print the answer

Below is the implementation of the above approach:

# Python program to count all possible paths from

# top left to top bottom right using

# Recursive Dynamic Programming

# Returns count of possible paths to reach

# cell at row number m and column number n from

# the topmost leftmost cell (cell at 1, 1)

**def** numberOfPaths(n, m, DP):

**if** (n **==** 1 **or** m **==** 1):

        DP[n][m] **=** 1

**return** 1

    # Add the element in the DP table

    # If it was not computed before

**if** (DP[n][m] **==** 0):

        DP[n][m] **=** numberOfPaths(n **-** 1, m, DP) **+** numberOfPaths(n, m **-** 1, DP)

**return** DP[n][m]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    # Create an empty 2D table

    DP **=** [[0 **for** i **in** range(4)] **for** j **in** range(4)]

**print**(numberOfPaths(3, 3, DP))

# This code is contributed by gauravrajput1

**Output**

6

**Time Complexity:**O(N \* M)

**Auxiliary Space:**(N \* M)

**34.Count all possible paths from the top left to the bottom right of a M X N matrix using**[DP](https://www.geeksforgeeks.org/dynamic-programming/)**:**

To solve the problem follow the below idea:

*So this problem has both properties (see*[*this*](https://www.geeksforgeeks.org/dynamic-programming-set-1/)*and*[*this*](https://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/)*) of a dynamic programming problem. Like other typical*[*Dynamic Programming(DP) problems*](https://www.geeksforgeeks.org/archives/tag/dynamic-programming)*, recomputations of the same subproblems can be avoided by constructing a temporary array count[][] in a bottom-up manner using the above recursive formula*

Follow the below steps to solve the problem:

* Declare a 2-D array count of size M \* N
* Set value of count[i][0] equal to 1 for 0 <= i < M as the answer of subproblem with a single column is equal to 1
* Set value of count[0][j] equal to 1 for 0 <= j < N as the answer of subproblem with a single row is equal to 1
* Create a nested for loop for 0 <= i < M and 0 <= j < N and assign count[i][j] equal to count[i-1][j] + count[i][j-1]
* Print value of count[M-1][N-1]

Below is the implementation of the above approach:

# Python3 program to count all possible paths

# from top left to bottom right

# Returns count of possible paths to reach cell

# at row number m and column number n from the

# topmost leftmost cell (cell at 1, 1)

**def** numberOfPaths(m, n):

    # Create a 2D table to store

    # results of subproblems

    # one-liner logic to take input for rows and columns

    # mat = [[int(input()) for x in range (C)] for y in range(R)]

    count **=** [[0 **for** x **in** range(n)] **for** y **in** range(m)]

    # Count of paths to reach any

    # cell in first column is 1

**for** i **in** range(m):

        count[i][0] **=** 1

    # Count of paths to reach any

    # cell in first row is 1

**for** j **in** range(n):

        count[0][j] **=** 1

    # Calculate count of paths for other

    # cells in bottom-up

    # manner using the recursive solution

**for** i **in** range(1, m):

**for** j **in** range(1, n):

            count[i][j] **=** count[i**-**1][j] **+** count[i][j**-**1]

**return** count[m**-**1][n**-**1]

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

  m **=** 3

  n **=** 3

  print(numberOfPaths(m, n))

# This code is contributed by Aditi Sharma

**Output**

6

**Time Complexity:** O(M \* N) – Due to nested for loops.

**Auxiliary Space:** O(M \* N) – We have used a 2D array of size M x N

**Space optimization of the above approach:**

To solve the problem follow the below idea:

*We can space optimize the above dp approach as for calculating the values of the current row we require only previous row*

Follow the below steps to solve the problem:

* Declare an array dp of size N
* Set dp[0] = 1
* Create a nested for loop for 0 <= i < M and 0 <= j < N and add dp[j-1] to dp[j]
* Print value of dp[n – 1]

Below is the implementation of the above approach:

# Returns count of possible paths

# to reach cell at row number m and

# column number n from the topmost

# leftmost cell (cell at 1, 1)

**def** numberOfPaths(p, q):

    # Create a 1D array to store

    # results of subproblems

    dp **=** [1 **for** i **in** range(q)]

**for** i **in** range(p **-** 1):

**for** j **in** range(1, q):

            dp[j] **+=** dp[j **-** 1]

**return** dp[q **-** 1]

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

  print(numberOfPaths(3, 3))

# This code is contributed

# by Ankit Yadav

**Output**

6

**Time Complexity:**O(M \* N)

**Auxiliary Space:**O(N)

This code is contributed by [**Vivek Singh**](https://www.facebook.com/vvkvivek.singh)

**Note:** the count can also be calculated using the formula (M-1 + N-1)!/(M-1)! \* (N-1)!

**35.Count all possible paths from top left to the bottom right of a M X N matrix using**[combinatorics](https://www.geeksforgeeks.org/combinatorics-gq/)**:**

To solve the problem follow the below idea:

*In this approach, We have to calculate m+n-2Cn-1 here which will be (m+n-2)! / (n-1)! (m-1)!*

*m = number of rows, n = number of columns*

*Total number of moves in which we have to move down to reach the last row = m – 1 (m rows, since we are starting from (1, 1) that is not included)*

*Total number of moves in which we have to move right to reach the last column = n – 1 (n column, since we are starting from (1, 1) that is not included)*

*Down moves = (m – 1)*

*Right moves = (n – 1)*

*Total moves = Down moves + Right moves = (m – 1) + (n – 1)*

*Now think of moves as a string of ‘R’ and ‘D’ characters where ‘R’ at any ith index will tell us to move ‘Right’ and ‘D’ will tell us to move ‘Down’. Now think of how many unique strings (moves) we can make where in total there should be (n – 1 + m – 1) characters and there should be (m – 1) ‘D’ character and (n – 1) ‘R’ character?*

*Choosing positions of (n – 1) ‘R’ characters results in the automatic choosing of (m – 1) ‘D’ character positions*

*The number of ways to choose positions for (n – 1) ‘R’ character = Total positions C n – 1 = Total positions C m – 1 = (n – 1 + m – 1) !=*

***Another way to think about this problem:***

*Count the Number of ways to make an N digit Binary String (String with 0s and 1s only) with****‘X’ zeros****and****‘Y’ ones****(here we have replaced ‘R’ with ‘0’ or ‘1’ and ‘D’ with ‘1’ or ‘0’ respectively whichever suits you better)*

Follow the below steps to solve the problem:

* Declare a variable path equal to 1
* Create a for loop from i equal to n to (m + n – 1)
* Set path equal to path \* i
* Set path equal to path divided by (i – n + 1)
* Return path

Below is the implementation of the above approach:

# Python3 program to count all possible

# paths from top left to top bottom

# using combinatorics

**def** numberOfPaths(m, n):

    path **=** 1

    # We have to calculate m + n-2 C n-1 here

    # which will be (m + n-2)! / (n-1)! (m-1)! path = 1;

**for** i **in** range(n, (m **+** n **-** 1)):

        path **\*=** i

        path **//=** (i **-** n **+** 1)

**return** path

# Driver code

print(numberOfPaths(3, 3))

# This code is contributed

# by Akanksha Rai

**Output**

6

**Time Complexity:**O(M)

**Auxiliary Space:**O(1)

**36.Unique paths in a Grid with Obstacles**

Given a grid of size m \* n, let us assume you are starting at (1, 1) and your goal is to reach (m, n). At any instance, if you are on (x, y), you can either go to (x, y + 1) or (x + 1, y).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space are marked as 1 and 0 respectively in the grid.

**Examples:**

Input: [[0, 0, 0],  
 [0, 1, 0],  
 [0, 0, 0]]  
Output : 2  
There is only one obstacle in the middle.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Method 1:**Recursion

We have discussed the problem to count the [number of unique paths in a Grid](https://www.geeksforgeeks.org/print-all-possible-paths-from-top-left-to-bottom-right-of-a-mxn-matrix/) when no obstacle was present in the grid. But here the situation is quite different. While moving through the grid, we can get some obstacles that we can not jump and the way to reach the bottom right corner is blocked.

# Python code to find number of unique paths

# in a Matrix

**def**  UniquePathHelper(i, j, r, c, A):

    # boundary condition or constraints

**if**(i **==** r **or** j **==** c):

**return** 0

**if**(A[i][j] **==** 1):

**return** 0

    # base case

**if**(i **==** r**-**1 **and** j **==** c**-**1):

**return** 1

**return**  UniquePathHelper(i**+**1, j, r, c, A) **+** UniquePathHelper(i, j**+**1, r, c, A)

**def** uniquePathsWithObstacles(A):

    r,c **=** len(A),len(A[0])

**return** UniquePathHelper(0, 0, r, c, A)

# Driver code

A **=** [ [ 0, 0, 0 ],

      [ 0, 1, 0 ],

      [ 0, 0, 0 ] ]

print(uniquePathsWithObstacles(A))

# This code is contributed by shinjanpatra

**Output**

2

**Time Complexity:** O(2m\*n)

**Auxiliary Space:** O(m\*n)

**Method 2:**Using DP

**1) Top-Down**

The most efficient solution to this problem can be achieved using dynamic programming. Like every dynamic problem concept, we will not recompute the subproblems. A temporary 2D matrix will be constructed and value will be stored using the top-down approach.

# Python code to find number of unique paths

# in a Matrix

**def** UniquePathHelper(i, j, r, c, A, paths):

    # boundary condition or constraints

**if** (i **==** r **or** j **==** c):

**return** 0

**if** (A[i][j] **==** 1):

**return** 0

    # base case

**if** (i **==** r **-** 1 **and** j **==** c **-** 1):

**return** 1

**if** (paths[i][j] !**= -**1):

**return** paths[i][j]

    paths[i][j] **=** UniquePathHelper(

        i **+** 1, j, r, c, A, paths) **+** UniquePathHelper(i, j **+** 1, r, c, A, paths)

**return** paths[i][j]

**def** uniquePathsWithObstacles(A):

    r, c **=** len(A), len(A[0])

    # create a 2D-matrix and initializing

    # with value 0

    paths **=** [[**-**1 **for** i **in** range(c)]**for** j **in** range(r)]

**return** UniquePathHelper(0, 0, r, c, A, paths)

# Driver code

A **=** [[0, 0, 0], [0, 1, 0], [0, 0, 0]]

print(uniquePathsWithObstacles(A))

# code is contributed by shinjanpatra

**Output**

2

**Time Complexity:**O(m\*n)

**Auxiliary Space:** O(m\*n)

**2) Bottom-Up**

A temporary 2D matrix will be constructed and value will be stored using the bottom-up approach.

**Approach:**

* Create a 2D matrix of the same size as the given matrix to store the results.
* Traverse through the created array row-wise and start filling the values in it.
* If an obstacle is found, set the value to 0.
* For the first row and column, set the value to 1 if an obstacle is not found.
* Set the sum of the right and the upper values if an obstacle is not present at that corresponding position in the given matrix
* Return the last value of the created 2d matrix

Below is the implementation of the above approach:

# Python code to find number of unique paths in a

# matrix with obstacles.

**def** uniquePathsWithObstacles(A):

    # create a 2D-matrix and initializing with value 0

    paths **=** [[0]**\***len(A[0]) **for** i **in** A]

    # initializing the left corner if no obstacle there

**if** A[0][0] **==** 0:

        paths[0][0] **=** 1

    # initializing first column of the 2D matrix

**for** i **in** range(1, len(A)):

        # If not obstacle

**if** A[i][0] **==** 0:

            paths[i][0] **=** paths[i**-**1][0]

    # initializing first row of the 2D matrix

**for** j **in** range(1, len(A[0])):

        # If not obstacle

**if** A[0][j] **==** 0:

            paths[0][j] **=** paths[0][j**-**1]

**for** i **in** range(1, len(A)):

**for** j **in** range(1, len(A[0])):

            # If current cell is not obstacle

**if** A[i][j] **==** 0:

                paths[i][j] **=** paths[i**-**1][j] **+** paths[i][j**-**1]

    # returning the corner value of the matrix

**return** paths[**-**1][**-**1]

# Driver Code

A **=** [[0, 0, 0], [0, 1, 0], [0, 0, 0]]

print(uniquePathsWithObstacles(A))

**Output**

2

**Time Complexity:**O(m\*n)

**Auxiliary Space:** O(m\*n)

**Method 3:** Space Optimization of DP solution.

In this method, we will use the given ‘A’ 2D matrix to store the previous answer using the bottom-up approach.

**Approach**

* Start traversing through the given ‘A’ 2D matrix row-wise and fill the values in it.
* For the first row and the first column set the value to 1 if an obstacle is not found.
* For the first row and first column, if an obstacle is found then start filling 0 till the last index in that particular row or column.
* Now start traversing from the second row and column ( eg: A[ 1 ][ 1 ]).
* If an obstacle is found, set 0 at particular Grid ( eg: A[ i ][ j ] ), otherwise set sum of upper and left values at A[ i ][ j ].
* Return the last value of the 2D matrix.

Below is the implementation of the above approach.

# Python program for the above approach

**def** uniquePathsWithObstacles(A):

    r **=** len(A)

    c **=** len(A[0])

    # If obstacle is at starting position

**if** (A[0][0]):

**return** 0

    #  Initializing starting position

    A[0][0] **=** 1

    # first row all are '1' until obstacle

**for** j **in** range(1,c):

**if** (A[0][j] **==** 0):

            A[0][j] **=** A[0][j **-** 1]

**else**:

            # No ways to reach at this index

            A[0][j] **=** 0

    # first column all are '1' until obstacle

**for** i **in** range(1,r):

**if** (A[i][0] **==** 0):

            A[i][0] **=** A[i **-** 1][0]

**else**:

            # No ways to reach at this index

            A[i][0] **=** 0

**for** i **in** range(1,r):

**for** j **in** range(1,c):

            # If current cell has no obstacle

**if** (A[i][j] **==** 0):

                A[i][j] **=** A[i **-** 1][j] **+** A[i][j **-** 1]

**else**:

                # No ways to reach at this index

                A[i][j] **=** 0

    # returning the bottom right

    # corner of Grid

**return** A[r **-** 1]

# Driver Code

A **=** [ [ 0, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 0 ] ]

**print**(uniquePathsWithObstacles(A))

# This code is contributed by shinjanpatra

**Output**

2

**Time Complexity:** O(m\*n)

**Auxiliary Space:** O(1)

**The 2D Dp Approach:**

As Per Problem tell us that we can move in two ways  can either go to (x, y + 1) or (x + 1, y). So we just calculate all possible outcome in both ways and store in 2d dp vector and return the dp[0][0] i.e all possible ways that takes you from (0,0) to (n-1,m-1);

# Python code for the above approach

**def** uniquePathsWithObstacles(grid):

    n **=** len(grid)

    m **=** len(grid[0])

**if** n **==** 1 **and** m **==** 1 **and** grid[0][0] **==** 0:

**return** 1

**if** n **==** 1 **and** m **==** 1 **and** grid[0][0] **==** 1:

**return** 0

    dp **=** [[**-**1 **for** j **in** range(m)] **for** i **in** range(n)]

**def** path(dp, grid, i, j):

**if** i < n **and** j < m **and** grid[i][j] **==** 1:

**return** 0

**if** i **==** n **-** 1 **and** j **==** m **-** 1:

**return** 1

**if** i >**=** n **or** j >**=** m:

**return** 0

**if** dp[i][j] !**= -**1:

**return** dp[i][j]

        left **=** path(dp, grid, i **+** 1, j)

        right **=** path(dp, grid, i, j **+** 1)

        dp[i][j] **=** left **+** right

**return** dp[i][j]

    path(dp, grid, 0, 0)

**if** dp[0][0] **== -**1:

**return** 0

**return** dp[0][0]

# Driver Code

grid **=** [[0, 0, 0], [0, 1, 0], [0, 0, 0]]

print(uniquePathsWithObstacles(grid))

# This code is contributed by lokeshpotta20.

**Output**

2

**Time Complexity**: O(M\*N),For traversing all possible ways.

**Auxiliary Space**: O(M\*N),For storing in 2D Dp Vector.