

# Tabulation vs Memoization - GeeksforGeeks

**Source:** <https://www.geeksforgeeks.org/tabulation-vs-memoization/>

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund

**Tabulation vs Memoization** Last Updated : 3 Feb, 2026

Tabulation and memoization are two techniques used to implement dynamic programming. Both techniques are used when there are overlapping subproblems (the same subproblem is executed multiple times). Below is an overview of two approaches.

**Memoization:** Top-down approach Stores the results of function calls in a table. Recursive implementation Entries are filled when needed.

**Tabulation:** Bottom-up approach Stores the results of subproblems in a table Iterative implementation Entries are filled in a bottom-up manner from the smallest size to the final size.

**Memoization Tabulation State State Transition** relation is easy to think State transition relation is difficult to think Code Code is easy to write by modifying the underlying recursive solution. Code gets complicated when a lot of conditions are required Speed Slow due to a lot of recursive calls. Fast, as we do not have recursion call overhead. Subproblem solving If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required If all subproblems must be solved at least once, a bottom-up dynamic programming algorithm definitely outperforms a top-down memoized algorithm by a constant factor Table entries Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand. In the Tabulated version, starting from the first entry, all entries are filled one by one

**Implementation Analysis:**

**Rod Cutting Problem** Given a rod of length  $n$  inches and an array `price[]`. `price[i]` denotes the value of a piece of length  $i$ . The task is to determine the maximum value obtainable by cutting up the rod and selling the pieces. Examples: Input: `price[] = [1, 5, 8, 9, 10, 17, 17, 20]` Output: 22 Explanation : The maximum obtainable value is 22 by cutting in two pieces of lengths 2 and 6, i.e.,  $5 + 17 = 22$ . Input : `price[] = [3, 5, 8, 9, 10, 17, 17, 20]` Output : 24 Explanation : The maximum obtainable value is 24 by cutting the rod into 8 pieces of length 1, i.e,  $8 * price[1] = 8 * 3 = 24$ . Input : `price[] = [3]` Output : 3 Explanation: There is only 1 way to pick a piece of length 1. In the rod cutting problem, the goal is to determine the maximum profit that can be obtained by cutting a rod into smaller pieces and selling them, given a price list for each possible piece length. The approach involves considering all possible cuts for the rod and recursively calculating the maximum profit for each cut. For detailed explanation and approaches, refer to Rod Cutting . Using Top-Down DP (Memoization) -  $O(n^2)$  Time and  $O(n)$  Space In this implementation of the rod cutting problem, memoization is used to optimize the recursive approach by storing the results of subproblems, avoiding redundant calculations.

**C++** // C++ program to find maximum // profit from rod of size  $n$

```
#include <bits/stdc++.h> using namespace std ; int cutRodRecur ( int i , vector < int > & price , vector < int > & memo ) { // Base case if ( i == 0 ) return 0 ; // If value is memoized if ( memo [ i - 1 ] != -1 ) return memo [ i - 1 ] ; int ans = 0 ; // Find maximum value for each cut. // Take value of rod of length j, and // recursively find value of rod of // length (i-j). for ( int j = 1 ; j <= i ; j ++ ) { ans = max ( ans , price [ j - 1 ] + cutRodRecur ( i - j , price , memo ) ) ; } return memo [ i - 1 ] = ans ; } int cutRod ( vector < int > & price ) { int n = price . size () ; vector < int > memo ( price . size () , -1 ) ; return cutRodRecur ( n , price , memo ) ; } int main () { vector < int > price = { 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 } ; cout << cutRod ( price ) ; return 0 ; }
```

**Java** // Java program to find maximum // profit from rod of size  $n$

```
import java.util.* ; class GfG { static int cutRodRecur ( int i , int [] price , int [] memo ) { // Base case if ( i == 0 ) return 0 ; // If value is memoized if ( memo [ i - 1 ] != -1 ) return memo [ i - 1 ] ; int ans = 0 ; // Find maximum value for each cut. // Take value of rod of length j, and // recursively find value of rod of // length (i-j). for ( int j = 1 ; j <= i ; j ++ ) { ans = Math . max ( ans , price [ j - 1 ] +
```

```

cutRodRecur ( i - j , price , memo )); } return memo [ i - 1 ] = ans ; } static int cutRod ( int [] price ) { int n
= price . length ; int [] memo = new int [ n ] ; Arrays . fill ( memo , - 1 ) ; return cutRodRecur ( n , price ,
memo ) ; } public static void main ( String [] args ) { int [] price = { 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 } ; System .
out . println ( cutRod ( price )) ; } } Python # Python program to find maximum # profit from rod of size n
def cutRodRecur ( i , price , memo ) : # Base case if i == 0 : return 0 # If value is memoized if memo [ i -
1 ] != - 1 : return memo [ i - 1 ] ans = 0 # Find maximum value for each cut. # Take value of rod of length
j , and # recursively find value of rod of # length (i-j). for j in range ( 1 , i + 1 ) : ans = max ( ans , price [ j -
1 ] + cutRodRecur ( i - j , price , memo )) memo [ i - 1 ] = ans return ans def cutRod ( price ) : n = len (
price ) memo = [ - 1 ] * n return cutRodRecur ( n , price , memo ) if __name__ == "__main__" : price = [
1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 ] print ( cutRod ( price )) C# // C# program to find maximum // profit from
rod of size n using System ; class GfG { static int cutRodRecur ( int i , int [] price , int [] memo ) { // Base
case if ( i == 0 ) return 0 ; // If value is memoized if ( memo [ i - 1 ] != - 1 ) return memo [ i - 1 ] ; int ans =
0 ; // Find maximum value for each cut. // Take value of rod of length j , and // recursively find value of
rod of // length (i-j). for ( int j = 1 ; j <= i ; j ++ ) { ans = Math . Max ( ans , price [ j - 1 ] + cutRodRecur ( i -
j , price , memo )) ; } return memo [ i - 1 ] = ans ; } static int cutRod ( int [] price ) { int n = price . Length ;
int [] memo = new int [ n ] ; Array . Fill ( memo , - 1 ) ; return cutRodRecur ( n , price , memo ) ; } static
void Main ( string [] args ) { int [] price = { 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 } ; Console . WriteLine ( cutRod (
price )) ; } } JavaScript // JavaScript program to find maximum // profit from rod of size n function
cutRodRecur ( i , price , memo ) { // Base case if ( i === 0 ) return 0 ; // If value is memoized if ( memo [ i
- 1 ] !== - 1 ) return memo [ i - 1 ] ; let ans = 0 ; // Find maximum value for each cut. // Take value of rod
of length j , and // recursively find value of rod of // length (i-j). for ( let j = 1 ; j <= i ; j ++ ) { ans = Math .
max ( ans , price [ j - 1 ] + cutRodRecur ( i - j , price , memo )) ; } memo [ i - 1 ] = ans ; return ans ; }
function cutRod ( price ) { const n = price . length ; const memo = Array ( n ) . fill ( - 1 ) ; return
cutRodRecur ( n , price , memo ) ; } const price = [ 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 ] ; console . log ( cutRod
( price )) ; Output 22 Using Bottom-Up DP (Tabulation) - O(n^2) Time and O(n) Space We iteratively
calculate the maximum profit for each possible rod length. For each length i , we check all possible
smaller cuts, update the profit by comparing the current maximum profit with the profit obtained by
combining smaller cuts, and ultimately return the maximum profit for the entire rod. C++ // C++ program
to find maximum // profit from rod of size n #include <bits/stdc++.h> using namespace std ; int cutRod (
vector < int > & price ) { int n = price . size () ; vector < int > dp ( price . size () + 1 , 0 ) ; // Find maximum
value for all // rod of length i. for ( int i = 1 ; i <= n ; i ++ ) { for ( int j = 1 ; j <= i ; j ++ ) { dp [ i ] = max ( dp [
i ] , price [ j - 1 ] + dp [ i - j ] ) ; } } return dp [ n ] ; } int main () { vector < int > price = { 1 , 5 , 8 , 9 , 10 , 17 ,
17 , 20 } ; cout << cutRod ( price ) ; return 0 ; } Java // Java program to find maximum // profit from rod of
size n import java.util.* ; class GfG { static int cutRod ( int [] price ) { int n = price . length ; int [] dp = new
int [ n + 1 ] ; // Find maximum value for all // rod of length i. for ( int i = 1 ; i <= n ; i ++ ) { for ( int j = 1 ; j
<= i ; j ++ ) { dp [ i ] = Math . max ( dp [ i ] , price [ j - 1 ] + dp [ i - j ] ) ; } } return dp [ n ] ; } public static
void main ( String [] args ) { int [] price = { 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 } ; System . out . println ( cutRod (
price )) ; } } Python # Python program to find maximum # profit from rod of size n def cutRod ( price ) : n
= len ( price ) dp = [ 0 ] * ( n + 1 ) # Find maximum value for all # rod of length i. for i in range ( 1 , n + 1
) : for j in range ( 1 , i + 1 ) : dp [ i ] = max ( dp [ i ] , price [ j - 1 ] + dp [ i - j ] ) return dp [ n ] if __name__ ==
 "__main__" : price = [ 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 ] print ( cutRod ( price )) C# // C# program to find
maximum // profit from rod of size n using System ; class GfG { static int cutRod ( int [] price ) { int n =
price . Length ; int [] dp = new int [ n + 1 ] ; // Find maximum value for all // rod of length i. for ( int i = 1 ; i
<= n ; i ++ ) { for ( int j = 1 ; j <= i ; j ++ ) { dp [ i ] = Math . Max ( dp [ i ] , price [ j - 1 ] + dp [ i - j ] ) ; } } return
dp [ n ] ; } static void Main ( string [] args ) { int [] price = { 1 , 5 , 8 , 9 , 10 , 17 , 17 , 20 } ; Console .
WriteLine ( cutRod ( price )) ; } } JavaScript // JavaScript program to find maximum // profit from rod of
size n function cutRod ( price ) { const n = price . length ; const dp = Array ( n + 1 ) . fill ( 0 ) ; // Find
maximum value for all // rod of length i. for ( let i = 1 ; i <= n ; i ++ ) { for ( let j = 1 ; j <= i ; j ++ ) { dp [ i ] =
Math . max ( dp [ i ] , price [ j - 1 ] + dp [ i - j ] ) ; } } return dp [ n ] ; } const price = [ 1 , 5 , 8 , 9 , 10 , 17 , 17 ,
20 ] ; console . log ( cutRod ( price )) ; Output 22 Comment Article Tags: Article Tags: Dynamic
Programming DSA

```