**Count number of ways to reach destination in a Maze**

Given a maze with obstacles, count number of paths to reach rightmost-bottommost cell from topmost-leftmost cell. A cell in given maze has value -1 if it is a blockage or dead end, else 0. From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only.

Input: maze[R][C] = {{0, 0, 0, 0},

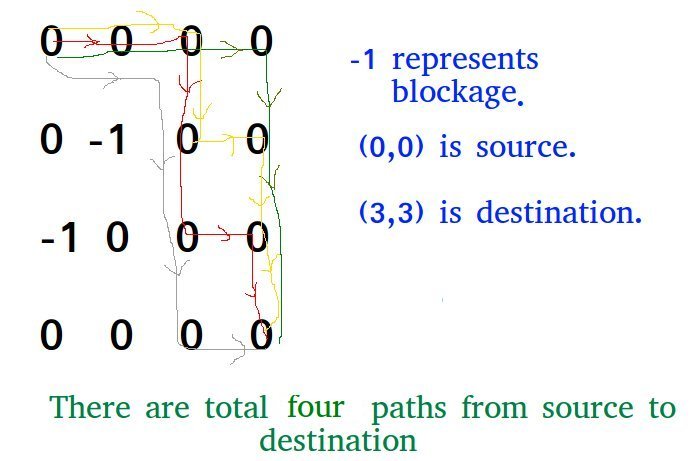
{0, -1, 0, 0},

{-1, 0, 0, 0},

{0, 0, 0, 0}};

Output: 4

There are four possible paths as shown in below diagram.



This problem is an extension of below problem.

[**Backtracking | Set 2 (Rat in a Maze)**](http://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/)

In this post a different solution is discussed that can be used to solve the above Rat in a Maze problem also.

The idea is to modify the given grid[][] so that **grid[i][j] contains count of paths to reach (i, j) from (0, 0) if (i, j) is not a blockage, else grid[i][j] remains -1.**

We can recursively compute grid[i][j] using below

formula and finally return grid[R-1][C-1]

// If current cell is a blockage

if (maze[i][j] == -1)

maze[i][j] = -1; // Do not change

// If we can reach maze[i][j] from maze[i-1][j]

// then increment count.

else if (maze[i-1][j] > 0)

maze[i][j] = (maze[i][j] + maze[i-1][j]);

// If we can reach maze[i][j] from maze[i][j-1]

// then increment count.

else if (maze[i][j-1] > 0)

maze[i][j] = (maze[i][j] + maze[i][j-1]);

public class CountNoWaysToReachDestination {

public static void main(String[] args) {

int maze[][] = new int[][]{{0, 0, 0, 0},

{0, -1, 0, 0},

{-1, 0, 0, 0},

{0, 0, 0, 0}};

System.out.println(countWays(maze,maze.length));

}

private static int countWays(int[][] maze, int n) {

int dp[][] = new int[n][n];

dp[0][0] = 1;

// Initializing the leftmost column and // Similarly initialize the topmost row

for(int i = 1;i<n;i++){

dp[i][0] = maze[i][0]!=-1? dp[i-1][0]:0;

dp[0][i] = maze[0][i]!=-1? dp[0][i-1]:0;

}

for(int i=1;i<n;i++){

for(int j=1;j<n;j++){

if(maze[i][j]==-1)

dp[i][j] = 0;

else

dp[i][j] = dp[i-1][j] + dp[i][j-1];

}

}

return dp[n-1][n-1];

}

}

**Print All the path from Starting to End.**

public class PrintPath {

static void printPaths (String tempString, int i, int j, int m, int n, char [][] arr) {

String newString = tempString + arr[i][j];

if (i == m -1 && j == n-1) {

System.out.println(newString);

return;

}

//right

if (j+1 < n) {

printPaths (newString, i, j+1, m, n, arr);

}

//down

if (i+1 < m) {

printPaths (newString, i+1, j, m, n, arr);

}

}

public static void main (String[] args) throws java.lang.Exception

{

char [] [] pathInput = {

{'1', '2', '3'},

{'4', '5', '6'},

};

printPaths ("", 0, 0, pathInput.length, pathInput[0].length, pathInput);

}

}

# Edit Distance:- Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’ .

# Insert

# Remove

# Replace

# All of the above operations are of equal cost.

# Input: str1 = "geek", str2 = "gesek"

# Output: 1

# We can convert str1 into str2 by inserting a 's'.

# Input: str1 = "cat", str2 = "cut"

# Output: 1

# We can convert str1 into str2 by replacing 'a' with 'u'.

# Input: str1 = "sunday", str2 = "saturday"

# Output: 3

# Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.

# Replace 'n' with 'r', insert t, insert a

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

**m:** Length of str1 (first string)

**n:** Length of str2 (second string)

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

// A Naive recursive Java program to find minimum number

// operations to convert str1 to str2

class EDIST

{

    static int min(int x,int y,int z)

    {

        if (x<y && x<z) return x;

        if (y<x && y<z) return y;

        else return z;

    }

    static int editDist(String str1 , String str2 , int m ,int n)

    {

        // If first string is empty, the only option is to

    // insert all characters of second string into first

    if (m == 0) return n;

    // If second string is empty, the only option is to

    // remove all characters of first string

    if (n == 0) return m;

    /\* If last characters of two strings are same, nothing

    much to do. Ignore last characters and get count for remaining strings.\*/

    if (str1.charAt(m-1) == str2.charAt(n-1))

        return editDist(str1, str2, m-1, n-1);

    /\* If last characters are not same, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values \*/

    return 1 + min (editDist(str1,  str2, m, n-1),    // Insert

                    editDist(str1,  str2, m-1, n),   // Remove

                  editDist(str1,  str2, m-1, n-1) // Replace

                   );

    }

    public static void main(String args[])

    {

        String str1 = "sunday";

        String str2 = "saturday";

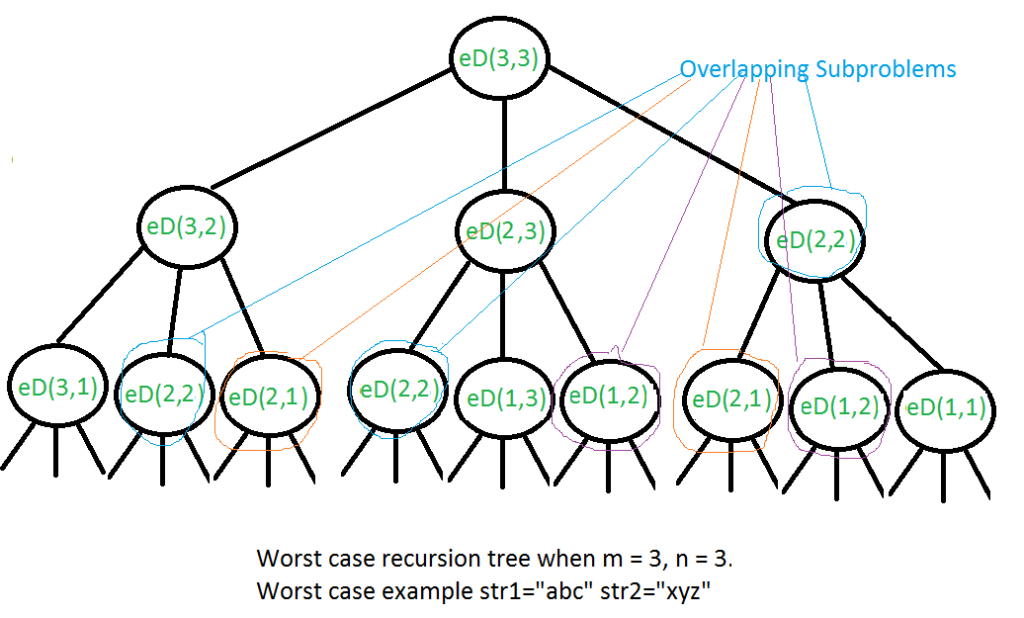
        System.out.println( editDist( str1 , str2 , str1.length(), str2.length()) );

    }

}

Output:- 3

The time complexity of above solution is exponential. In worst case, we may end up doing O(3m) operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



We can see that many subproblems are solved again and again, for example eD(2,2) is called three times. Since same suproblems are called again, this problem has Overlapping Subprolems property. So Edit Distance problem has both properties (see [this](http://www.geeksforgeeks.org/archives/12635)and [this](http://www.geeksforgeeks.org/archives/12819)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subpriblems.

|  |
| --- |
| // A Dynamic Programming based Java program to find minimum  // number operations to convert str1 to str2  class EDIST  {      static int min(int x,int y,int z)      {          if (x < y && x <z) return x;          if (y < x && y < z) return y;          else return z;      }        static int editDistDP(String str1, String str2, int m, int n)      {          // Create a table to store results of subproblems          int dp[][] = new int[m+1][n+1];            // Fill d[][] in bottom up manner          for (int i=0; i<=m; i++)          {              for (int j=0; j<=n; j++)              {                  // If first string is empty, only option is to                  // isnert all characters of second string                  if (i==0)                      dp[i][j] = j;  // Min. operations = j                    // If second string is empty, only option is to                  // remove all characters of second string                  else if (j==0)                      dp[i][j] = i; // Min. operations = i                    // If last characters are same, ignore last char                  // and recur for remaining string                  else if (str1.charAt(i-1) == str2.charAt(j-1))                      dp[i][j] = dp[i-1][j-1];                    // If last character are different, consider all                  // possibilities and find minimum                  else                      dp[i][j] = 1 + min(dp[i][j-1],  // Insert                                         dp[i-1][j],  // Remove                                         dp[i-1][j-1]); // Replace              }          }            return dp[m][n];      }            public static void main(String args[])      {          String str1 = "sunday";          String str2 = "saturday";          System.out.println( editDistDP( str1 , str2 , str1.length(), str2.length()) );      }  } |

**Output:-3**

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

**Find minimum number of coins that make a given value :-** Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = { C1, C2, .. , Cm} valued coins, what is the minimum number of coins to make the change?

**Examples:-** Input: coins[] = {25, 10, 5}, V = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, V = 11

Output: Minimum 2 coins required

We can use one coin of 6 cents and 1 coin of 5 cents

This problem is a variation of the problem discussed [Coin Change Problem](http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/). Here instead of finding total number of possible solutions, we need to find the solution with minimum number of coins.

The minimum number of coins for a value V can be computed using below recursive formula.

If V == 0, then 0 coins required.

If V > 0

minCoin(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}

where i varies from 0 to m-1

and coin[i] <= V

//A Naive recursive JAVA program to find minimum of coins to make a given change V

class coin

{

    // m is size of coins array (number of different coins)

    static int minCoins(int coins[], int m, int V)

    {

       // base case

       if (V == 0) return 0;

       // Initialize result

       int res = Integer.MAX\_VALUE;

       // Try every coin that has smaller value than V

       for (int i=0; i<m; i++)

       {

         if (coins[i] <= V)

         {

             int sub\_res = minCoins(coins, m, V-coins[i]);

        //Check for INT\_MAX to avoid overflow and see if result can minimized

             if (sub\_res != Integer.MAX\_VALUE && sub\_res + 1 < res)

                res = sub\_res + 1;

         }

       }

       return res;

    }

    public static void main(String args[])

    {

       int coins[] =  {9, 6, 5, 1};

       int m = coins.length;

       int V = 11;

       System.out.println("Minimum coins required is "+ minCoins(coins, m, V) );

    }

}

**Output**:- Minimum coins required is 2

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observer that many subproblems are solved again and again. For example, when we start from V = 11, we can reach 6 by subtracting one 5 times and by subtracting 5 one times. So the subproblem for 6 is called twice.

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

public class CoinChangingMinimumCoin {

public int minimumCoinBottomUp(int total, int coins[]){

int T[] = new int[total + 1];

int R[] = new int[total + 1];

T[0] = 0;

for(int i=1; i <= total; i++){

T[i] = Integer.MAX\_VALUE-1;

R[i] = -1;

}

for(int j=0; j < coins.length; j++){

for(int i=1; i <= total; i++){

if(i >= coins[j]){

if (T[i - coins[j]] + 1 < T[i]) {

T[i] = 1 + T[i - coins[j]];

R[i] = j;

}

}

}

}

printCoinCombination(R, coins);

return T[total];

}

private void printCoinCombination(int R[], int coins[]) {

if (R[R.length - 1] == -1) {

System.out.print("No solution is possible");

return;

}

int start = R.length - 1;

System.out.print("Coins used to form total ");

while ( start != 0 ) {

int j = R[start];

System.out.print(coins[j] + " ");

start = start - coins[j];

}

System.out.print("\n");

}

public static void main ( String args[] ) {

int total = 13;

int coins[] = {7, 3, 2, 6};

CoinChangingMinimumCoin cc = new CoinChangingMinimumCoin();

int bottomUpValue = cc.minimumCoinBottomUp(total, coins);

System.out.print(String.format("Bottom up and top down result %s %s", bottomUpValue, topDownValue));

}

}

### **Asked in:** [Accolite](http://practice.geeksforgeeks.org/company/Accolite/),[Amazon](http://practice.geeksforgeeks.org/company/Amazon/),[Morgan-Stanley](http://practice.geeksforgeeks.org/company/Morgan-Stanley/),[Oracle](http://practice.geeksforgeeks.org/company/Oracle/),[Paytm](http://practice.geeksforgeeks.org/company/Paytm/),[Snapdeal](http://practice.geeksforgeeks.org/company/Snapdeal/),[Synopsys](http://practice.geeksforgeeks.org/company/Synopsys/)

**Coin Change: -** Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

**1) Optimal Substructure :-** To count total number solutions, we can divide all set solutions in two sets.  
1) Solutions that do not contain mth coin (or Sm).  
2) Solutions that contain at least one Sm.  
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

**2) Overlapping Subproblems :-** Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

/ Returns the count of ways we can sum  S[0...m-1] coins to get sum n

int count( int S[], int m, int n )

{

    // If n is 0 then there is 1 solution (do not include any coin)

    if (n == 0)

        return 1;

    // If n is less than 0 then no solution exists

    if (n < 0)

        return 0;

    // If there are no coins and n is greater than 0, then no solution exist

    if (m <=0 && n >= 1)

        return 0;

    // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]

    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );

}

// Driver program to test above function

int main()

{

    int i, j;

    int arr[] = {1, 2, 3};

    int m = sizeof(arr)/sizeof(arr[0]);

    printf("%d ", count(arr, m, 4));

    getchar();

    return 0;

}

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {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)

/

/

. .

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

public class CoinChanging {

public int numberOfSolutions(int total, int coins[]){

int temp[][] = new int[coins.length+1][total+1];

for(int i=0; i <= coins.length; i++){

temp[i][0] = 1;

}

for(int i=1; i <= coins.length; i++){

for(int j=1; j <= total ; j++){

if(coins[i-1] > j){

temp[i][j] = temp[i-1][j];

}

else{

temp[i][j] = temp[i][j-coins[i-1]] + temp[i-1][j];

}

}

}

return temp[coins.length][total];

}

public static void main(String args[]){

CoinChanging cc = new CoinChanging();

int total = 15;

int coins[] = {3,4,6,7,9};

System.out.println(cc.numberOfSolutions(total, coins));

}

}

/\* Dynamic Programming Java implementation of Coin Change problem \*/

class CoinChange

{

    static long countWays(int S[], int m, int n)

    {

        //Time complexity of this function: O(mn)

        //Space Complexity of this function: O(n)

/\* table[i] will be storing the number of solutions for value i.We need n+1 rows as the table is constructed in bottom up manner using base case(n = 0)

        long[] table = new long[n+1];

        // Initialize all table values as 0

        Arrays.fill(table, 0);   //O(n)

        // 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 (int i=0; i<m; i++)

            for (int j=S[i]; j<=n; j++)

                table[j] += table[j-S[i]];

        return table[n];

    }

    public static void main(String args[])

    {

        int arr[] = {1, 2, 3};

        int m = arr.length;

        int n = 4;

        System.out.println(countWays(arr, m, n));

    }

}

# Subset Sum Problem:- 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.

# Examples: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

# Output: True //There is a subset (4, 5) with sum 9.

# Let isSubSetSum(int set[], int n, int sum) be the function to find whether there is a subset of set[] with sum equal to *sum*. n is the number of elements in set[].

# The isSubsetSum problem can be divided into two subproblems …a) Include the last element, recur for n = n-1, sum = sum – set[n-1] …b) Exclude the last element, recur for n = n-1. If any of the above the above subproblems return true, then return true.

# 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

// A recursive solution for subset sum problem

class subset\_sum

{

    // Returns true if there is a subset of set[] with sum

        // equal to given sum

    static boolean isSubsetSum(int set[], int n, int sum)

    {

       // Base Cases

       if (sum == 0)

         return true;

       if (n == 0 && sum != 0)

         return false;

       // If last element is greater than sum, then ignore it

       if (set[n-1] > sum)

         return isSubsetSum(set, n-1, sum);

       /\* else, check if sum can be obtained by any of the following

          (a) including the last element (b) excluding the last element   \*/

       return isSubsetSum(set, n-1, sum) ||

                                   isSubsetSum(set, n-1, sum-set[n-1]);

    }

    /\* Driver program to test above function \*/

    public static void main (String args[])

    {

          int set[] = {3, 34, 4, 12, 5, 2};

          int sum = 9;

          int n = set.length;

          if (isSubsetSum(set, n, sum) == true)

             System.out.println("Found a subset with given sum");

          else

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

    }

}

**Output:-**  Found a subset with given sum

# 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).

# We can solve the problem in [Pseudo-polynomial time](http://en.wikipedia.org/wiki/Pseudo-polynomial_time) using Dynamic programming. We create a boolean 2D table subset[][] and fill it in bottom up manner. The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false. Finally, we return subset[sum][n]