**[LOVE FOR PROGRAMMING](https://loveforprogramming.quora.com/)**

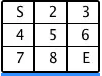
<https://loveforprogramming.quora.com/Backtracking-Memoization-Dynamic-Programming>

**Backtracking, Memoization & Dynamic Programming!**

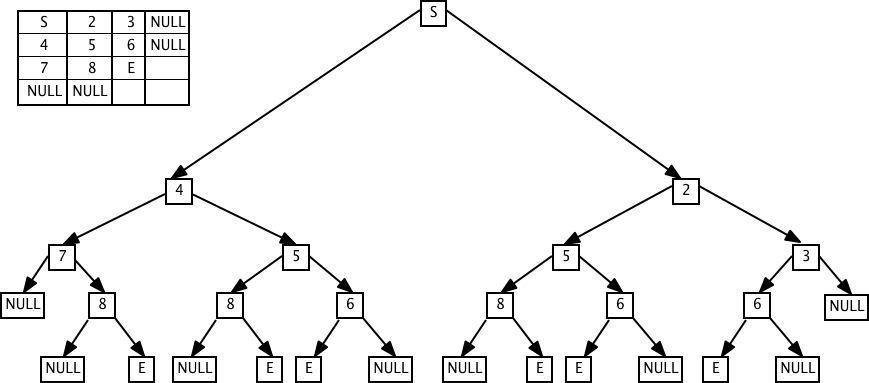
[Pawan Bhadauria](https://www.quora.com/profile/Pawan-Bhadauria)

[Backtracking](http://en.wikipedia.org/wiki/Backtracking) is a fundamental concept essential to solve many problems in computer science. Many NP-hard problems require use of backtracking. Almost all problems, which require use of backtracking are inherently recursive in nature. You will encounter many problems, specially in graph theory, which require backtracking. If not impossible, it would be really hard to solve those problems without recursion. Though recursion is a powerful concept, it also brings in its own set of challenges as the input size becomes larger. In many recursive backtracking programs, there are huge number of recursive calls made for same set of input values again and again. To optimize this, we can cache the solution to recursive calls for specific input values and reuse those if we encounter those input values again. This concept of remembering and reuse of the solution for a specific set of input values is called [Memoization](http://en.wikipedia.org/wiki/Memoization). This technique of using memoization for optimizing programs using backtracking is nothing but [Dynamic programming](http://en.wikipedia.org/wiki/Dynamic_Programming).

Enough theory!! Lets discuss this with the help of a classic problem. Lets say we are given below matrix, and we need to find number of routes from S to E (S & E are integers are well). The constraint is that we are allowed to only move down or move right.

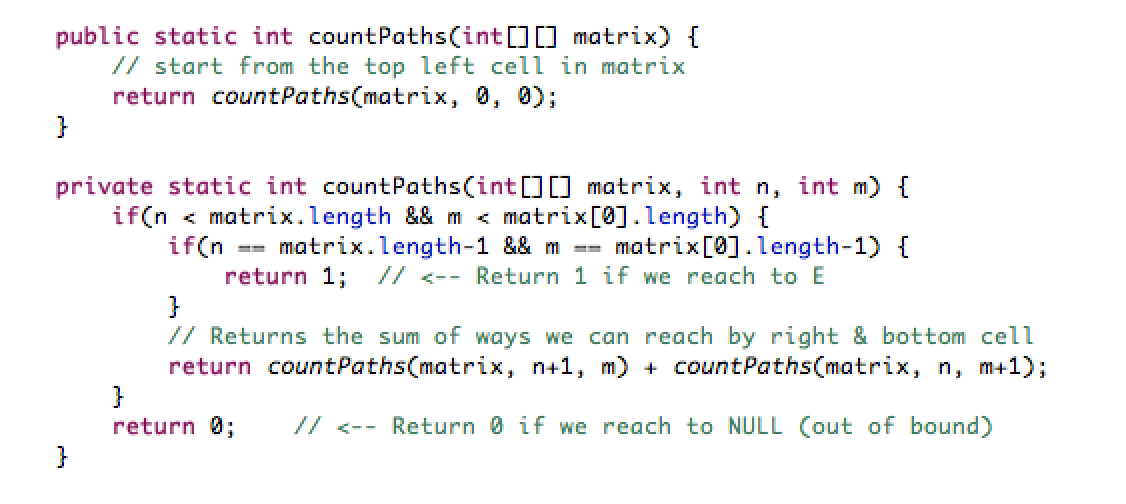


This is fairly simple to solve if you are well versed with recursion and backtracking. Anyways, lets try to formulate a solution. Since at every cell, we can either move right or down, this sort of translates into a binary decision tree. If we draw each flow, through a particular cell, as a separate branch, it might look something like this. [In the matrix, we added NULL for cells which are out of bound]



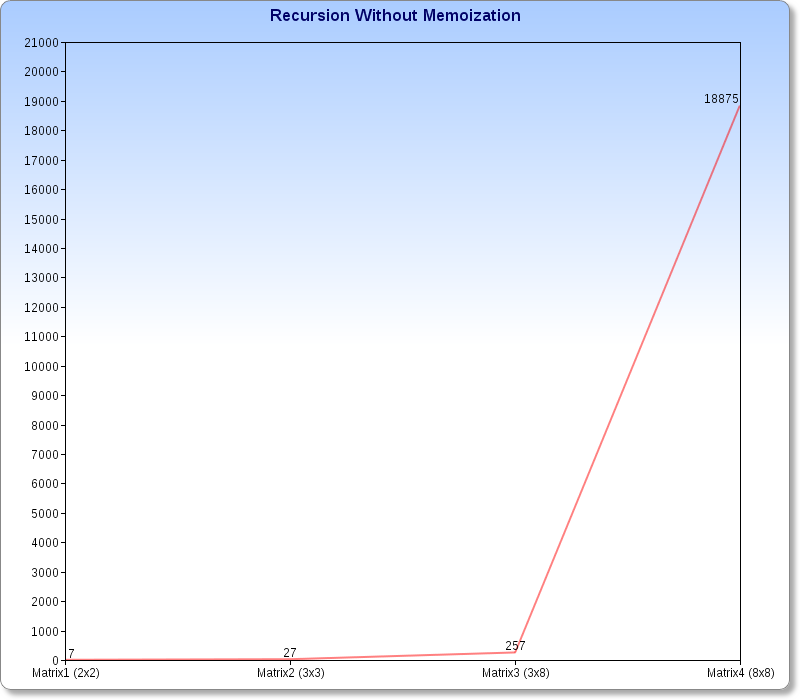
Having got the flow in the form of a tree, this can easily be traversed using recursion. If we had intermingled common nodes across different branches, we would have got a Graph instead and doing a DFS on it would have helped us. But lets get back to the binary tree format of the flow for simplicity.

If we were to write a recursive solution to the problem, it would come out to be something like this (in Java):

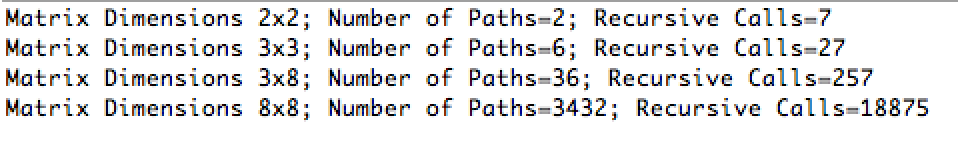


Based on above tree, this solution essentially traverses to every leaf in the tree. If the leaf is NULL (out of bound cell), there is no path (0) and if leaf is 'E', there is a path (1). At every node, it adds the number of routes offered by left & right child. It finally returns the total routes for parent node. For example, at '6', there is 1 path but at '5', there are 2 paths, from '6' and '8' (1+1). This works great and returns "6" as number of routes from "S" to "E".

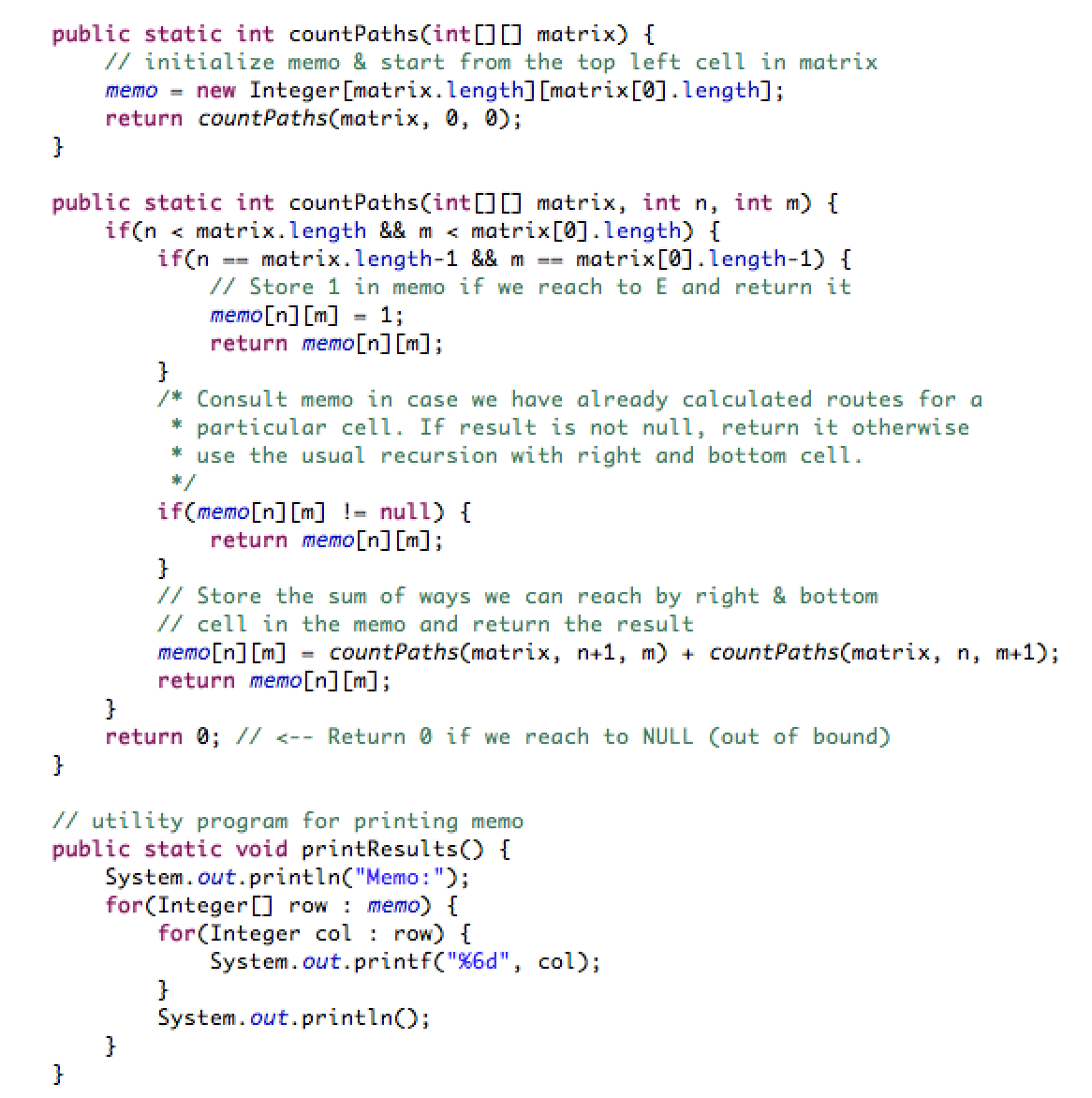
Do you see any problem with this solution? There is one. Every flow is independent of each other. So if solution traverses from S->2->5 & calculates the routes for 5, it will do this calculation again when it traverses to 5 via S->4. I ran this solution for different input matrices & below is the number of times 'countPaths' method was recursively called.

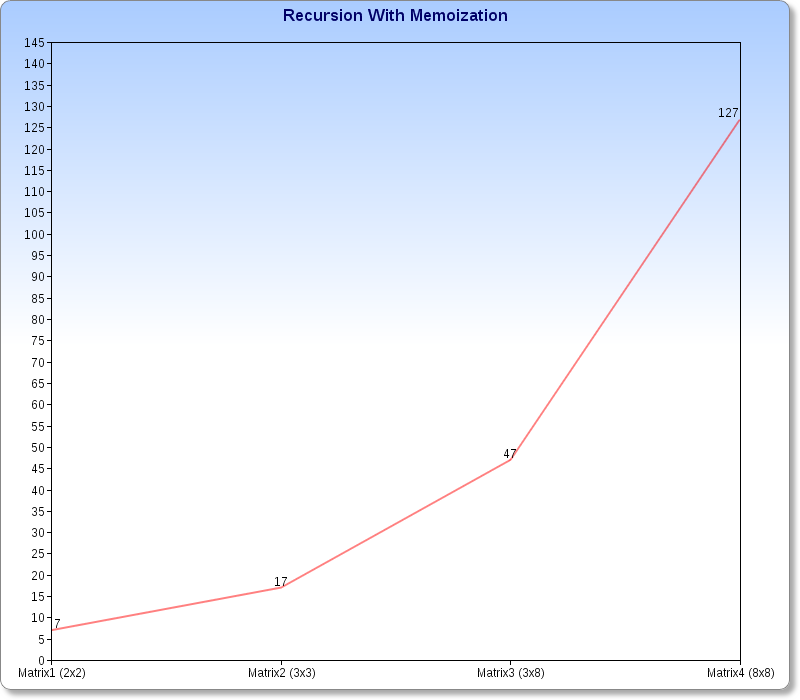


The number of calls for a 8x8 matrix is astoundingly high and as the dimensions of input matrix tend to get larger, the number of recursive calls will increase exponentially. Clearly, this solution will not scale well, once the input size becomes even moderately high. Here is a more verbose output of the program for few matrices (use links at the end of the post to see the exact test matrices):

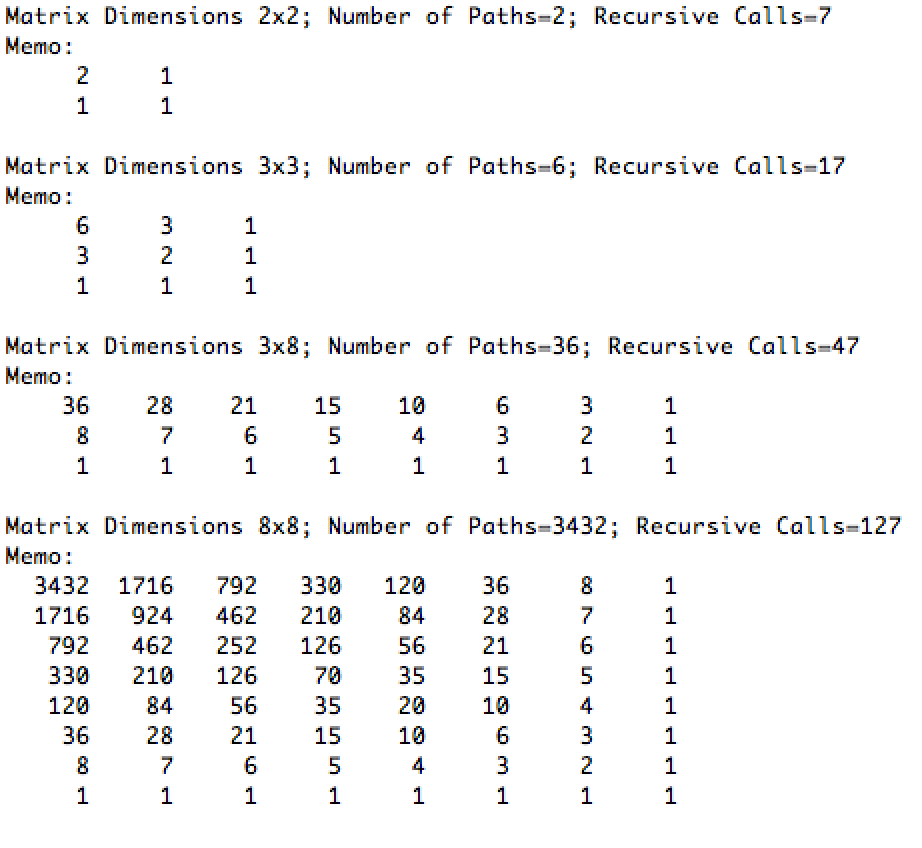


A better solution would be to cache the already calculated routes for a node (as it will not change) and reuse it if we encounter that node again. This leads us to the concept of memoization as explained above. A memoization is a simple technique to cache the result of a method call for a set of input(s) and reuse it if we see those inputs again. Lets try to modify the solution based on memoization.

As you can see above, we have created a 'memo' matrix to store the number of routes for each cell once it is calculated, which is subsequently reused during execution, thats it. Below is the recursive call count for 'countPaths' method, with this modified version, now:



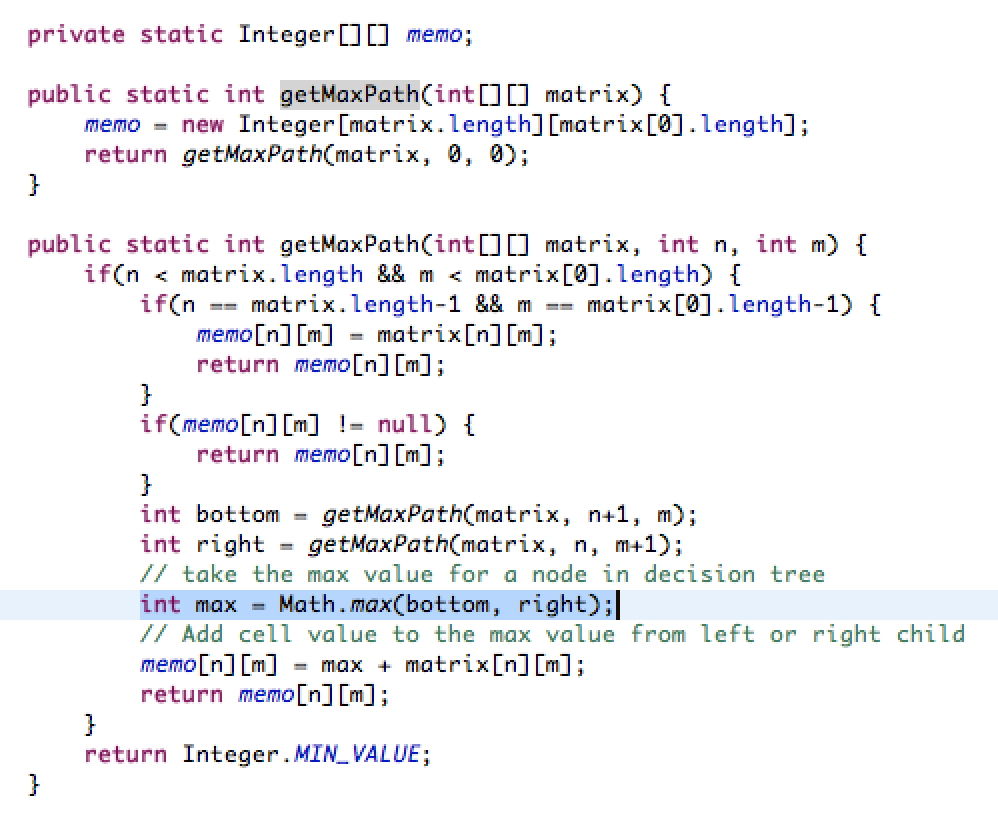
As you can see, an astonishing reduction in the number of recursive calls. ***For 8x8 matrix, it reduces from 18875 to 127***. Here is a more verbose program output for the same set of matrices, along with formatted output of 'memo' matrix with intermediate results (use links at the end of the post to see the exact test matrices):



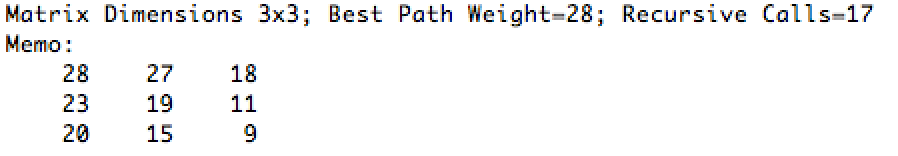
To extend this a little further, lets say we are given below matrix with each cell representing the value. We want to find, path from 1->9 which offers maximum value. The constraint is that we are allowed to only move down or move right.



This is an optimization problem (rather then reduction) but very similar to earlier problem. The above solution can be easily modified to calculate maximum value path. Here is a solution, with memoization, which calculates the maximum value path. *We modified the above solution to now recursively take maximum value out of left and right child & then add the value of current node to it. As usual, we also memoize the calls on each cell. Pretty simple!!*



If we run this program for provided matrix, we get below output:



This methodology can work wonders for large input sizes thus increasing the performance of program dramatically with memoization. ***This programming methodology, in simple terms, is Dynamic Programming, an optimization over backtracking***.

The [Knapsack problem](http://en.wikipedia.org/wiki/Knapsack_problem) (0/1), which is also an optimization problem, can be easily solved by following similar methodology (as illustrated above for maximum value path problem). Dynamic programming can't provide us an optimal algorithm for problems which are considered NP-hard but it can still help in increasing the efficiency of programs which deal with such problems.

***Solutions for both route and maximum value path problems can be found here***:

<https://github.com/pbhadauria/loveforprogramming/tree/master/java/com/lfp/dp>

***The code samples related to all my posts for this blog can be found on my github repository:***[***pbhadauria/loveforprogramming***](https://github.com/pbhadauria/loveforprogramming)

40,946 views · [222 upvotes](https://loveforprogramming.quora.com/Backtracking-Memoization-Dynamic-Programming) · [Posted Mar 22, 2014](https://loveforprogramming.quora.com/Backtracking-Memoization-Dynamic-Programming)