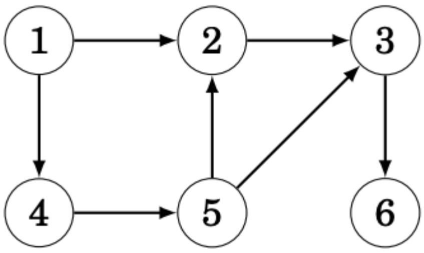
Total number of paths in a Directed Acyclic Graph

Problem Statement
You are given a Weighted Directed Acyclic Graph (DAG) consisting of ‘N’ nodes and ‘E’ directed edges. Your task is to find the total number of paths from node 1 to node N .

As an example, let us calculate the number of paths from node 1 to node 6 in the following graph:



There are a total of three such paths:
• 1 → 2 → 3 → 6
• 1 → 4 → 5 → 2 → 3 → 6
• 1 → 4 → 5 → 3 → 6

Brute Force
We start from the source node using a dfs call and visit every neighbour of the current node. If we reach node N, we simply increase the count of total paths to n by 1.

The implementation of the above algorithm is as follows:

```
/*
        Helper function to count all paths from current node to node N recursively.
        'count' is passed by reference.
*/

function countPathsHelper(current, adj, ref(count)):

        //  if the path ends at node N increment the count
        if current == N
                count += 1
                return

        //  Recursively try all possible paths from current.
```

```
        for v in adj[current]:
                countPathsHelper(v, adj, count)


/*
        Function to count all paths from node 1 to node N.
*/


function countPaths(N, adj):
        //   variable to store the count of paths
        count = 0


        countPathsHelper(1, adj, 0)
        return count
```

Time Complexity: O(N^N), where 'N' is the number of nodes in the given directed graph. For each node there are at most 'N' vertices that can be visited from the current node and there are 'N' nodes in the graph. Thus, overall it will take O(N^N) time to explore all the paths.

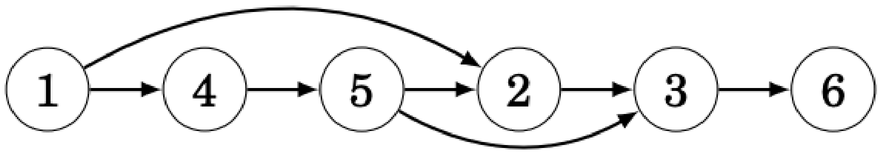
Space Complexity: O(N + E), where 'N' is the number of nodes in the given directed graph and E is the number of edges. The space used by the adjacency list 'adj' is of the order of O(N + E),the space used by the recursion stack is of order O(N). Thus, the final space complexity will be O(N+E) + O(N) = O(N+E).

Dynamic Programming

Let paths(x) denote the number of paths from node 1 to node x. As a base case, paths(1) = 1. Then, to calculate other values of paths(x), we may use the recursion

paths(x) = paths(a1) + paths(a2) + ⋯ + paths(ak)

where a1,a2,...,ak are the nodes from which there is an edge to x. Since the graph is acyclic, the values of paths(x) can be calculated in the order of a topological sort. A topological sort for the above graph is as follows:



For example, to calculate the value of paths(3), we can use the formula paths(2)+paths(5), because there are edges from nodes 2 and 5 to node 3. Since paths(2) = 2 and paths(5) = 1, we conclude that paths(3) = 3. We will store the value of paths(x) for a node x in the array ‘dp’.

Algorithm

- Create an adjacency list ‘adj’, such that adj[i][j] stores an integer v, representing that the ‘jth’ adjacent node of ‘i’ is node ‘v’.
- Create a list of integers 'dp'' of size 'N' and fill it with 0. We will use 1-based indexing.
- Create a list ‘order’ of size ‘N’ that will have topological sorting of the graph.
- Call the function topological_sort that will update the ‘order’ array and will contain the topological order of the nodes in the graph.,
- Assign ‘dp[1]’ := 1.
- Run a loop where ‘i’ ranges from 1 to ‘N’, and for each ‘i’ do following -:
    - Initialize an integer variable current = order[i].
    - Visit all the adjacent nodes 'neighbor’ of node current, and update their paths with dp[neighbor] = dp[neighbor] + dp[current].

```
/*
        Utility function to find the topological order of the graph
*/
function topologicalSortUtil(current, adj, visited, stk)

        //  Mark current node visited
        visited[current] = True

        //  Iterating over adjacent vertices.
        for v in adj[current]
                if(visited[v] == False)
                        topologicalSortUtil(v, adj, visited, stk)

        /*
                Push vertex in stack after pushing all its
                adjacent (and their adjacent and so on) vertices.
        */
        stk.append(current)
```

```
/*
        Function to find the topological order of the graph
*/
function topologicalSort(adj)
        //   Number of nodes in a graph.
        n = len(adj)

        visited = array[n]
        stk = []

        //   Recursively finding topological sorting.
        for i from 1 to n
                if(visited[i] == False)
                        topologicalSortUtil(i, adj, visited, stk)

        //   List 'result' will keep a topological sort of given graph.
        result = []
        while(len(stk) > 0)
                result.append(stk.top())
                stk.pop()

        return result


/*
        Function to count the total number of paths in the graph from 1 to N
*/
function countPaths(n, adj)

        //   Find topological sorting.
        order = topologicalSort(adj)
```

```
//   Create a list 'dp' of size 'N' and fill it with zeros.
dp = array(n, 0)
dp[1] = 1

//   Find total paths of each node from 1.
for i in range(n)
        current = order[i]
        for neighbor in adj[current]
                dp[neighbor] = dp[neighbor] + dp[current]

return dp[N]
```

Time Complexity: O(N + E), where ‘N’ is the number of nodes in the given directed graph and E is the number of edges. Topological sorting will take the time of the order of O(N+E), and also we iterate over neighbors of all the nodes which also takes time O(N + E). Thus, the final complexity will be O(N+E) + O(N+E) = O(N+E).

Space Complexity: O(N + E), where ‘N’ is the number of nodes in the given directed graph and E is the number of edges. The space used by the adjacency list ‘adj’ is of the order of O(N + E) and the space used by array ‘dp’ and stack ‘stk’ is of order O(N). Thus, the final space complexity will be O(N+E) + O(N) = O(N+E).