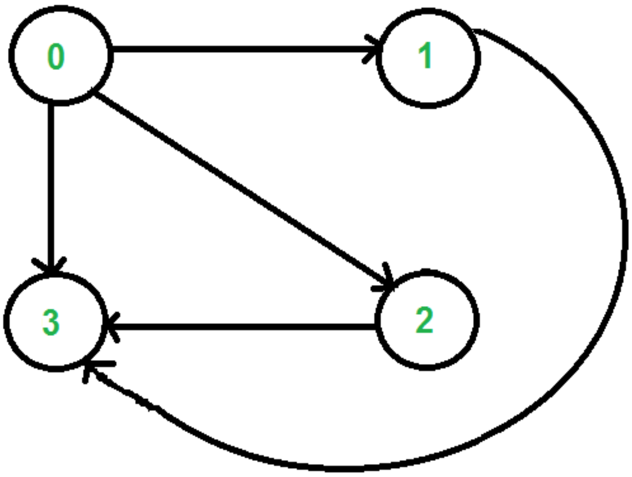
Given a directed graph and two vertices ‘u’ and ‘v’ in it, count all possible walks from ‘u’ to ‘v’ with exactly k edges on the walk.

The graph is given as adjacency matrix representation where value of graph[i][j] as 1 indicates that there is an edge from vertex i to vertex j and a value 0 indicates no edge from i to j.

For example consider the following graph. Let source ‘u’ be vertex 0, destination ‘v’ be 3 and k be 2. The output should be 2 as there are two walk from 0 to 3 with exactly 2 edges. The walks are {0, 2, 3} and {0, 1, 3}



We strongly recommend to minimize the browser and try this yourself first.

A **simple solution** is to start from u, go to all adjacent vertices and recur for adjacent vertices with k as k-1, source as adjacent vertex and destination as v. Following is C++ implementation of this simple solution.

```
// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A naive recursive function to count walks from u to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
   // Base cases
   if (k == 0 && u == v)      return 1;
   if (k == 1 && graph[u][v]) return 1;
   if (k <= 0)                return 0;

   // Initialize result
   int count = 0;

   // Go to all adjacents of u and recur
   for (int i = 0; i < V; i++)
       if (graph[u][i])  // Check if is adjacent of u
           count += countwalks(graph, i, v, k-1);

   return count;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
     int graph[V][V] = { {0, 1, 1, 1},
                         {0, 0, 0, 1},
                         {0, 0, 0, 1},
```

```
                        {0, 0, 0, 0}
                      };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Output:

```
2
```

The worst case time complexity of the above function is $O(V^k)$ where V is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly n children.
We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other Dynamic Programming problems, we fill the 3D table in bottom up manner.

```
// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A Dynamic programming based function to count walks from u
// to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value count[i][j][e] will
    // store count of possible walks from i to j with exactly k edges
    int count[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++)  // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                count[i][j][e] = 0;

                // from base cases
                if (e == 0 && i == j)
                    count[i][j][e] = 1;
                if (e == 1 && graph[i][j])
                    count[i][j][e] = 1;

                // go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++) // adjacent of source i
                        if (graph[i][a])
                            count[i][j][e] += count[a][j][e-1];
                }
           }
        }
    }
    return count[u][v][k];
}

// driver program to test above function
int main()
{
```

```
    /* Let us create the graph shown in above diagram*/
     int graph[V][V] = { {0, 1, 1, 1},
                         {0, 0, 0, 1},
                         {0, 0, 0, 1},
                         {0, 0, 0, 0}
                       };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}
```

Output:

```
2
```

Time complexity of the above DP based solution is $O(V^3K)$ which is much better than the naive solution.

We can also use **Divide and Conquer** to solve the above problem in $O(V^3Logk)$ time. The count of walks of length k from u to v is the [u][v]'th entry in $(graph[V][V])^k$. We can calculate power of by doing O(Logk) multiplication by using the divide and conquer technique to calculate power. A multiplication between two matrices of size V x V takes $O(V^3)$ time. Therefore overall time complexity of this method is $O(V^3Logk)$.