Graph Data Structure And Algorithms

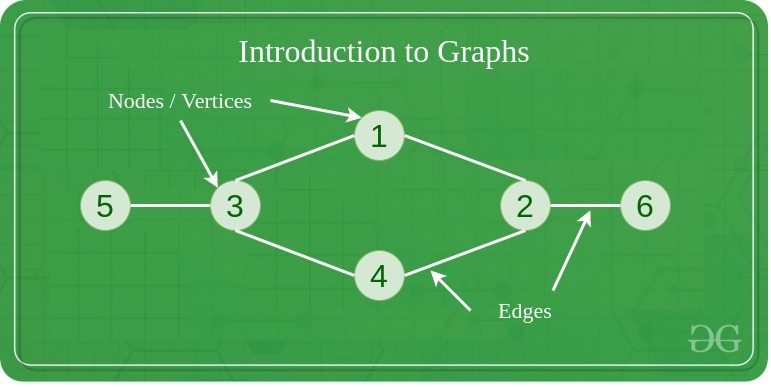
[**Data Structure and Algorithms Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=gfg&utm_medium=header+link+click&utm_campaign=dsa+course+tracker&utm_term=dsa+course+promo&utm_content=graph-lp)

[**Practice Problems on Graphs**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Graph&page=1&category%5B%5D=Graph&utm_source=gfg&utm_medium=header+link+click&utm_campaign=practice+tracker&utm_term=practice+promo&utm_content=graph-lp)

[**Recent Articles on Graph**](https://www.geeksforgeeks.org/category/graph/?utm_source=gfg&utm_medium=header+link+click&utm_campaign=recent+article+tracker&utm_term=recent+article+tracker&utm_content=graph-lp)

[**What is Graph Data Structure?**](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/)

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( V ) and a set of edges( E ). The graph is denoted by G(E, V).



**Components of a Graph**

* **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
* **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

**Topics**:

* [Introduction](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#introduction)
* [BFS & DFS in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#bfsndfs)
* [Cycles in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#cycle)
* [Shortest Paths in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#shortest)
* [Minimum Spanning Tree](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#MST)
* [Topological Sorting](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#topo)
* [Connectivity](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#connectivity)
* [Maximum Flow](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#maxiflow)
* [Some must do problems on Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#mustdo)
* [Quick Links](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/?ref=ghm#quickLinks)

**Introduction:**

1. [Introduction to Graphs](https://www.geeksforgeeks.org/introduction-to-graphs/)
2. [Graph and its representations](https://www.geeksforgeeks.org/graph-and-its-representations/)
3. [Types of Graphs with Examples](https://www.geeksforgeeks.org/graph-types-and-applications/)
4. [Basic Properties of a Graph](https://www.geeksforgeeks.org/basic-properties-of-a-graph/)
5. [Applications, Advantages and Disadvantages of Graph](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-graph/)
6. [Transpose graph](https://www.geeksforgeeks.org/transpose-graph/)
7. [Difference between graph and tree](https://www.geeksforgeeks.org/difference-between-graph-and-tree/)

**BFS and DFS in Graph:**

1. [Breadth First Traversal for a Graph](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)
2. [Depth First Traversal for a Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)
3. [Applications of Depth First Search](https://www.geeksforgeeks.org/applications-of-depth-first-search/)
4. [Applications of Breadth First Traversal](https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/)
5. [Iterative Depth First Search](https://www.geeksforgeeks.org/iterative-depth-first-traversal/)
6. [BFS for Disconnected Graph](https://www.geeksforgeeks.org/bfs-disconnected-graph/)
7. [Transitive Closure of a Graph using DFS](https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/)
8. [Difference between BFS and DFS](https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/)

**Cycles in Graph:**

1. [Detect Cycle in a Directed Graph](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/)
2. [Detect cycle in an undirected graph](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/)
3. [Detect cycle in a direct graph using colors](https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/)
4. [Detect a negative cycle in a Graph | (Bellman Ford)](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/)
5. [Cycles of length n in an undirected and connected graph](https://www.geeksforgeeks.org/cycles-of-length-n-in-an-undirected-and-connected-graph/)
6. [Detecting negative cycle using Floyd Warshall](https://www.geeksforgeeks.org/detecting-negative-cycle-using-floyd-warshall/)
7. [Clone a Directed Acyclic Graph](https://www.geeksforgeeks.org/clone-directed-acyclic-graph/)
8. [Union By Rank and Path Compression in Union-Find Algorithm](https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/)
9. [Introduction to Disjoint Set Data Structure or Union-Find Algorithm](https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/)

**Shortest Path in Graph:**

1. [Dijkstra’s shortest path algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
2. [Bellman–Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
3. [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
4. [Johnson’s algorithm for All-pairs shortest paths](https://www.geeksforgeeks.org/johnsons-algorithm/)
5. [Shortest Path in Directed Acyclic Graph](https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/)
6. [Dial’s Algorithm](https://www.geeksforgeeks.org/dials-algorithm-optimized-dijkstra-for-small-range-weights/)
7. [Multistage Graph (Shortest Path)](https://www.geeksforgeeks.org/multistage-graph-shortest-path/)
8. [Shortest path in an unweighted graph](https://www.geeksforgeeks.org/shortest-path-unweighted-graph/)
9. [Karp’s minimum mean (or average) weight cycle algorithm](https://www.geeksforgeeks.org/karps-minimum-mean-average-weight-cycle-algorithm/)
10. [0-1 BFS (Shortest Path in a Binary Weight Graph)](https://www.geeksforgeeks.org/0-1-bfs-shortest-path-binary-graph/)
11. [Find minimum weight cycle in an undirected graph](https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/)

**Minimum Spanning Tree:**

1. [Prim’s Minimum Spanning Tree (MST))](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/)
2. [Kruskal’s Minimum Spanning Tree Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)
3. [Difference between Prim’s and Kruskal’s algorithm for MST](https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/)
4. [Applications of Minimum Spanning Tree Problem](https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/)
5. [Minimum cost to connect all cities](https://www.geeksforgeeks.org/minimum-cost-connect-cities/)
6. [Total number of Spanning Trees in a Graph](https://www.geeksforgeeks.org/total-number-spanning-trees-graph/)
7. [Minimum Product Spanning Tree](https://www.geeksforgeeks.org/minimum-product-spanning-tree/)
8. [Reverse Delete Algorithm for Minimum Spanning Tree](https://www.geeksforgeeks.org/reverse-delete-algorithm-minimum-spanning-tree/)
9. [Boruvka’s algorithm for Minimum Spanning Tree](https://www.geeksforgeeks.org/greedy-algorithms-set-9-boruvkas-algorithm/)

**Topological Sorting:**

1. [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/)
2. [All topological sorts of a Directed Acyclic Graph](https://www.geeksforgeeks.org/all-topological-sorts-of-a-directed-acyclic-graph/)
3. [Kahn’s Algorithm for Topological Sorting](https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/)
4. [Maximum edges that can be added to DAG so that is remains DAG](https://www.geeksforgeeks.org/maximum-edges-can-added-dag-remains-dag/)
5. [Longest Path in a Directed Acyclic Graph](https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/)
6. [Topological Sort of a graph using departure time of vertex](https://www.geeksforgeeks.org/topological-sorting-using-departure-time-of-vertex/)

**Connectivity:**

1. [Articulation Points (or Cut Vertices) in a Graph](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/)
2. [Biconnected Components](https://www.geeksforgeeks.org/biconnected-components/)
3. [Bridges in a graph](https://www.geeksforgeeks.org/bridge-in-a-graph/)
4. [Eulerian path and circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/)
5. [Fleury’s Algorithm for printing Eulerian Path or Circuit](https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/)
6. [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)
7. [Count all possible walks from a source to a destination with exactly k edges](https://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/)
8. [Euler Circuit in a Directed Graph](https://www.geeksforgeeks.org/euler-circuit-directed-graph/)
9. [Length of shortest chain to reach the target word](https://www.geeksforgeeks.org/length-of-shortest-chain-to-reach-a-target-word/)
10. [Find if an array of strings can be chained to form a circle](https://www.geeksforgeeks.org/given-array-strings-find-strings-can-chained-form-circle/)
11. [Tarjan’s Algorithm to find strongly connected Components](https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/)
12. [Paths to travel each nodes using each edge (Seven Bridges of Königsberg)](https://www.geeksforgeeks.org/paths-travel-nodes-using-edgeseven-bridges-konigsberg/)
13. [Dynamic Connectivity | Set 1 (Incremental)](https://www.geeksforgeeks.org/dynamic-connectivity-set-1-incremental/)

**Maximum Flow**

1. [Max Flow Problem Introduction](https://www.geeksforgeeks.org/max-flow-problem-introduction/)
2. [Ford-Fulkerson Algorithm for Maximum Flow Problem](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)
3. [Find maximum number of edge disjoint paths between two vertices](https://www.geeksforgeeks.org/find-edge-disjoint-paths-two-vertices/)
4. [Find minimum s-t cut in a flow network](https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/)
5. [Maximum Bipartite Matching](https://www.geeksforgeeks.org/maximum-bipartite-matching/)
6. [Channel Assignment Problem](https://www.geeksforgeeks.org/channel-assignment-problem/)
7. [Introduction to Push Relabel Algorithm](https://www.geeksforgeeks.org/introduction-to-push-relabel-algorithm/)
8. [Karger’s Algorithm- Set 1- Introduction and Implementation](https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/)
9. [Dinic’s algorithm for Maximum Flow](https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/)

**Some must do Problems on Graph:**

1. [Find length of the largest region in Boolean Matrix](https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/)
2. [Count number of trees in a forest](https://www.geeksforgeeks.org/count-number-trees-forest/)
3. [A Peterson Graph Problem](https://www.geeksforgeeks.org/peterson-graph/)
4. [Clone an Undirected Graph](https://www.geeksforgeeks.org/clone-an-undirected-graph/)
5. [Graph Coloring (Introduction and Applications)](https://www.geeksforgeeks.org/graph-coloring-applications/)
6. [Traveling Salesman Problem (TSP) Implementation](https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/)
7. [Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)](https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/)
8. [K Centers Problem | Set 1 (Greedy Approximate Algorithm)](https://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/)
9. [Erdos Renyl Model (for generating Random Graphs)](https://www.geeksforgeeks.org/erdos-renyl-model-generating-random-graphs/)
10. [Chinese Postman or Route Inspection | Set 1 (introduction)](https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/)
11. [Hierholzer’s Algorithm for directed graph](https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/)
12. [Check whether a given graph is Bipartite or not](https://www.geeksforgeeks.org/bipartite-graph/)
13. [Snake and Ladder Problem](https://www.geeksforgeeks.org/snake-ladder-problem-2/)
14. [Boggle (Find all possible words in a board of characters)](https://www.geeksforgeeks.org/boggle-find-possible-words-board-characters/)
15. [Hopcroft Karp Algorithm for Maximum Matching-Introduction](https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/)
16. [Minimum Time to rot all oranges](https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/)
17. [Construct a graph from given degrees of all vertices](https://www.geeksforgeeks.org/construct-graph-given-degrees-vertices/)
18. [Determine whether a universal sink exists in a directed graph](https://www.geeksforgeeks.org/determine-whether-universal-sink-exists-directed-graph/)
19. [Number of sink nodes in a graph](https://www.geeksforgeeks.org/number-sink-nodes-graph/)
20. [Two Clique Problem (Check if Graph can be divided in two Cliques)](https://www.geeksforgeeks.org/two-clique-problem-check-graph-can-divided-two-cliques/)

**Quick Links :**

* [Top 10 Interview Questions on Depth First Search (DFS)](https://www.geeksforgeeks.org/top-10-interview-question-depth-first-search-dfs/)
* [Some interesting shortest path questions](https://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/)
* [Quizzes on Graph Traversal](https://www.geeksforgeeks.org/algorithms-gq/graph-traversals-gq/)
* [Quizzes on Graph Shortest Path](https://www.geeksforgeeks.org/algorithms-gq/graph-shortest-paths-gq/)
* [Quizzes on Graph Minimum Spanning Tree](https://www.geeksforgeeks.org/algorithms-gq/graph-minimum-spanning-tree-gq/)
* [Quizzes on Graphs](https://www.geeksforgeeks.org/data-structure-gq/graph-gq/)
* [Practice Problems on Graphs](https://practice.geeksforgeeks.org/topics/Graph/?ref=taocp)
* [Videos on Graphs](https://www.youtube.com/playlist?list=PLqM7alHXFySEaZgcg7uRYJFBnYMLti-nh)

**Some must do Problems on Graph:**

**Find size of the largest region in Boolean Matrix**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 22 Sep, 2022
* Read
* Discuss(40+)
* Courses
* Practice
* Video

Consider a matrix, where each cell contains either a ‘0’ or a ‘1’, and any cell containing a 1 is called a filled cell. Two cells are said to be connected if they are adjacent to each other horizontally, vertically, or diagonally. If one or more filled cells are also connected, they form a region. find the size of the largest region.

**Examples:**

***Input:****M[][5] = { {0, 0, 1, 1, 0}, {1, 0, 1, 1, 0}, {0, 1, 0, 0, 0}, {0, 0, 0, 0, 1}}*

***Output:****6*

***Explanation:****In the following example, there are 2 regions.*

*One with size 1 and the other as 6. So largest region: 6*

***Input:****M[][5] = { {0, 0, 1, 1, 0}, {0, 0, 1, 1, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0 1} }*

***Output:****4*

***Explanation:****In the following example, there are 2 regions.*

*One with size 1 and the other as 4. So largest region: 4*

Recommended Problem

Unit Area of largest region of 1's

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

[DFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=DFS&sortBy=submissions)

+3 more

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

[Flipkart](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Flipkart&sortBy=submissions)

+5 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/length-of-largest-region-of-1s-1587115620/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 53.4K

**Approach:** To solve the problem follow the below idea:

*The idea is based on the problem of*[*finding number of islands in Boolean 2D-matrix*](https://www.geeksforgeeks.org/find-number-of-islands/)

**Find the length of the largest region in Boolean Matrix using**[**DFS**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)**:**

Follow the given steps to solve the problem:

* A cell in the 2D matrix can be connected to at most 8 neighbors.
* So in DFS, make recursive calls for 8 neighbors of that cell.
* Keep a visited Hash-map to keep track of all visited cells.
* Also, keep track of the visited 1’s in every DFS and update the maximum size region.

Below is the implementation of the above approach:

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find the length of the

# largest region in boolean 2D-matrix

# A function to check if a given cell

# (row, col) can be included in DFS

**def** isSafe(M, row, col, visited):

**global** ROW, COL

    # row number is in range, column number is in

    # range and value is 1 and not yet visited

**return** ((row >**=** 0) **and** (row < ROW) **and**

            (col >**=** 0) **and** (col < COL) **and**

            (M[row][col] **and not** visited[row][col]))

# A utility function to do DFS for a 2D

# boolean matrix. It only considers

# the 8 neighbours as adjacent vertices

**def** DFS(M, row, col, visited, count):

    # These arrays are used to get row and column

    # numbers of 8 neighbours of a given cell

    rowNbr **=** [**-**1, **-**1, **-**1, 0, 0, 1, 1, 1]

    colNbr **=** [**-**1, 0, 1, **-**1, 1, **-**1, 0, 1]

    # Mark this cell as visited

    visited[row][col] **=** True

    # Recur for all connected neighbours

**for** k **in** range(8):

**if** (isSafe(M, row **+** rowNbr[k],

                   col **+** colNbr[k], visited)):

            # increment region length by one

            count[0] **+=** 1

            DFS(M, row **+** rowNbr[k],

                col **+** colNbr[k], visited, count)

# The main function that returns largest

# length region of a given boolean 2D matrix

**def** largestRegion(M):

**global** ROW, COL

    # Make a bool array to mark visited cells.

    # Initially all cells are unvisited

    visited **=** [[0] **\*** COL **for** i **in** range(ROW)]

    # Initialize result as 0 and traverse

    # through the all cells of given matrix

    result **= -**999999999999

**for** i **in** range(ROW):

**for** j **in** range(COL):

            # If a cell with value 1 is not

**if** (M[i][j] **and not** visited[i][j]):

                # visited yet, then new region found

                count **=** [1]

                DFS(M, i, j, visited, count)

                # maximum region

                result **=** max(result, count[0])

**return** result

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

  ROW **=** 4

  COL **=** 5

  M **=** [[0, 0, 1, 1, 0],

       [1, 0, 1, 1, 0],

       [0, 1, 0, 0, 0],

       [0, 0, 0, 0, 1]]

  # Function call

  print(largestRegion(M))

# This code is contributed by PranchalK

**Output**

6

**Time complexity:** O(ROW \* COL). In the worst case, all the cells will be visited so the time complexity is O(ROW \* COL).

**Auxiliary Space:**O(ROW \* COL). To store the visited nodes O(ROW \* COL) space is needed.

**Find the length of the largest region in Boolean Matrix using**[**BFS**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)**:**

Follow the given steps to solve the problem:

* If the value at any particular cell is 1 then from here we need to do the BFS traversal
* Push the pair<i,j> in the queue
* Marking the value 1 to -1 so that we don’t again push the same cell again
* We will check in all 8 directions and if we encounter the cell having a value of 1 then we will push it into the queue and we will mark the cell to -1

*From <*[*https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/*](https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/)*>*

**Count number of trees in a forest**

Given n nodes of a forest (collection of trees), find the number of trees in the forest.

**Examples :**

**Input :** edges[] = {0, 1}, {0, 2}, {3, 4}  
**Output :** 2  
**Explanation :** There are 2 trees  
 0 3  
 / \ \  
 1 2 4

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Approach :**

1. Apply DFS on every node.
2. Increment count by one if every connected node is visited from one source.
3. Again perform DFS traversal if some nodes yet not visited.
4. Count will give the number of trees in forest.

**Implementation:**

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to count number

# of trees in a forest.

# A utility function to add an

# edge in an undirected graph.

**def** addEdge(adj, u, v):

    adj[u].append(v)

    adj[v].append(u)

# A utility function to do DFS of graph

# recursively from a given vertex u.

**def** DFSUtil(u, adj, visited):

    visited[u] **=** True

**for** i **in** range(len(adj[u])):

**if** (visited[adj[u][i]] **==** False):

            DFSUtil(adj[u][i], adj, visited)

# Returns count of tree is the

# forest given as adjacency list.

**def** countTrees(adj, V):

    visited **=** [False] **\*** V

    res **=** 0

**for** u **in** range(V):

**if** (visited[u] **==** False):

            DFSUtil(u, adj, visited)

            res **+=** 1

**return** res

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    V **=** 5

    adj **=** [[] **for** i **in** range(V)]

    addEdge(adj, 0, 1)

    addEdge(adj, 0, 2)

    addEdge(adj, 3, 4)

    print(countTrees(adj, V))

# This code is contributed by PranchalK

**Output:**

2

**Time Complexity :** **O(V + E)**

*From <*[*https://www.geeksforgeeks.org/count-number-trees-forest/*](https://www.geeksforgeeks.org/count-number-trees-forest/)*>*

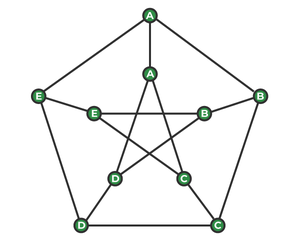
**A Peterson Graph Problem**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 25 Oct, 2022
* Read
* Discuss
* Courses
* Practice
* Video

The following graph G is called a Petersen graph and its vertices have been numbered from 0 to 9. Some letters have also been assigned to vertices of G, as can be seen from the following picture:

Let’s consider a walk W in graph G, which consists of L vertices W1, W2, …, WL. A string S of L letters ‘A’ – ‘E’ is realized by walking W if the sequence of letters written along W is equal to S. Vertices can be visited multiple times while walking along W.

For example, S = ‘ABBECCD’ is realized by W = (0, 1, 6, 9, 7, 2, 3). Determine whether there is a walk W that realizes a given string S in graph G and if so then find the lexicographically least such walk. The only line of input contains one string S. If there is no walk W which realizes S, then output -1 otherwise, you should output the least lexicographical walk W which realizes S.



*Example of a Petersen Graph*

**Examples:**

Input : s = 'ABB'  
Output: 016  
Explanation: As we can see in the graph  
 the path from ABB is 016.  
Input : s = 'AABE'  
Output :-1  
Explanation: As there is no path that  
 exists, hence output is -1.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Algorithm for a Peterson Graph Problem:**

**petersonGraphWalk(S, v):**

*begin*

*res := starting vertex*

*for each character c in S except the first one, do*

*if there is an edge between v and c in outer graph, then*

*v := c*

*else if there is an edge between v and c+5 in inner graph, then*

*v := c + 5*

*else*

*return false*

*end if*

*put v into res*

*done*

*return true*

*end*

Below is the implementation of the above algorithm:

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find the

# path in Peterson graph

# path to be checked

# adjacency matrix.

adj **=** [[False **for** i **in** range(10)] **for** j **in** range(10)]

# resulted path - way

result **=** [0]

# we are applying breadth first search

# here

**def** findthepath(S, v):

    result[0] **=** v

**for** i **in** range(1, len(S)):

        # first traverse the outer graph

**if** (adj[v][ord(S[i]) **-** ord('A')] **or**

            adj[ord(S[i]) **-** ord('A')][v]):

            v **=** ord(S[i]) **-** ord('A')

        # then traverse the inner graph

**else if** (adj[v][ord(S[i]) **-** ord('A') **+** 5] **or**

               adj[ord(S[i]) **-** ord('A') **+** 5][v]):

            v **=** ord(S[i]) **-** ord('A') **+** 5

        # if the condition failed to satisfy

        # return false

**else**:

**return** False

        result.append(v)

**return** True

# driver code

# here we have used adjacency matrix to make

# connections between the connected nodes

adj[0][1] **=** adj[1][2] **=** adj[2][3] **=** \

adj[3][4] **=** adj[4][0] **=** adj[0][5] **=** \

adj[1][6] **=** adj[2][7] **=** adj[3][8] **=** \

adj[4][9] **=** adj[5][7] **=** adj[7][9] **=** \

adj[9][6] **=** adj[6][8] **=** adj[8][5] **=** True

# path to be checked

S**=** "ABB"

S**=**list(S)

**if** (findthepath(S, ord(S[0]) **-** ord('A')) **or**

    findthepath(S, ord(S[0]) **-** ord('A') **+** 5)):

    print(**\***result, sep **=** "")

**else**:

**print**("-1")

# This code is contributed by SHUBHAMSINGH10

**Output**

016

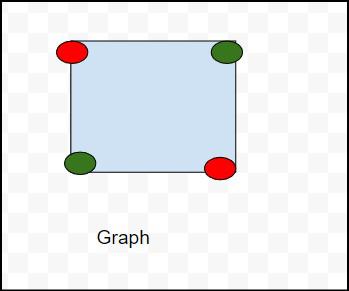
*From <*[*https://www.geeksforgeeks.org/peterson-graph/*](https://www.geeksforgeeks.org/peterson-graph/)*>*

**Graph Coloring | Set 1 (Introduction and Applications)**

[Graph coloring](http://en.wikipedia.org/wiki/Graph_coloring) problem is to assign colors to certain elements of a graph subject to certain constraints.

**Vertex coloring** is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like ***Edge Coloring*** (No vertex is incident to two edges of same color) and ***Face Coloring***(Geographical Map Coloring) can be transformed into vertex coloring.

**Chromatic Number:** The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 2 colors.



Chromatic number of this graph is 2 because  in this above diagram we can use to color red and green .

so chromatic number of this graph is 2 and is denoted x(G)  ,means x(G)=2 .

Chromatic number define as the least no of colors needed for coloring the graph .

and types of chromatic number are:

1) Cycle graph

2) planar graphs

3) Complete graphs

4) Bipartite Graphs:

5) Trees

The problem to find chromatic number of a given graph is [NP Complete](https://www.geeksforgeeks.org/np-completeness-set-1/).

**Applications of Graph Coloring:**

The graph coloring problem has huge number of applications.

***1) Making Schedule or Time Table:***Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

***2)***[***Mobile Radio Frequency Assignment***](http://www.zib.de/groetschel/teaching/SS2012/GraphCol%20and%20FrequAssignment.pdf)***:*** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

***3) Sudoku:***Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

***4)***[***Register Allocation***](http://en.wikipedia.org/wiki/Register_allocation)***:***In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

***5) Bipartite Graphs:***We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](https://www.geeksforgeeks.org/bipartite-graph/)for more details.

***6) Map Coloring:***Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](http://en.wikipedia.org/wiki/Four_color_theorem))

**There can be many more applications:** For example the below reference video lecture has a case study at 1:18.

*From <*[*https://www.geeksforgeeks.org/graph-coloring-applications/*](https://www.geeksforgeeks.org/graph-coloring-applications/)*>*

**Traveling Salesman Problem (TSP) Implementation**

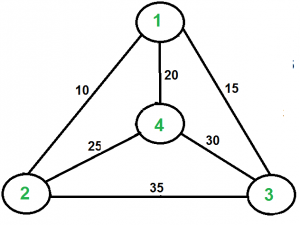
* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 20 Dec, 2022
* Read
* Discuss(20+)
* Courses
* Practice
* Video

[Travelling Salesman Problem (TSP) :](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.



**Examples:**

Output of Given Graph:  
minimum weight Hamiltonian Cycle :  
10 + 25 + 30 + 15 := 80

[Recommended: Please try your approach on ***{Practice}*** first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/travelling-salesman-problem2732/1)

In this post, the implementation of a simple solution is discussed.

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Below is the implementation of the above idea

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to implement traveling salesman

# problem using naive approach.

**from** sys **import** maxsize

**from** itertools **import** permutations

V **=** 4

# implementation of traveling Salesman Problem

**def** travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex **=** []

**for** i **in** range(V):

**if** i !**=** s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path **=** maxsize

    next\_permutation**=**permutations(vertex)

**for** i **in** next\_permutation:

        # store current Path weight(cost)

        current\_pathweight **=** 0

        # compute current path weight

        k **=** s

**for** j **in** i:

            current\_pathweight **+=** graph[k][j]

            k **=** j

        current\_pathweight **+=** graph[k][s]

        # update minimum

        min\_path **=** min(min\_path, current\_pathweight)

**return** min\_path

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # matrix representation of graph

    graph **=** [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

    s **=** 0

    print(travellingSalesmanProblem(graph, s))

**Output**

80

*From <*[*https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/*](https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/)*>*

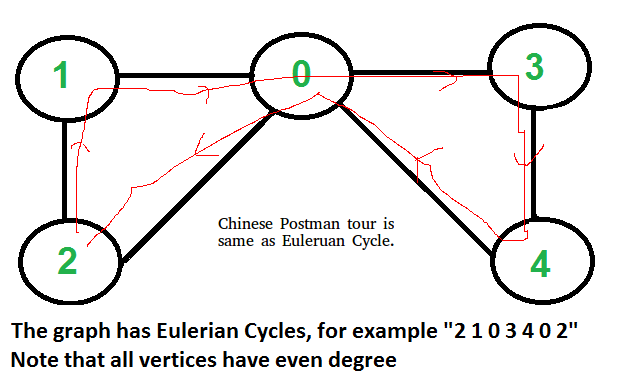
**Chinese Postman or Route Inspection | Set 1 (introduction)**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 07 Jul, 2022
* Read
* Discuss
* Courses
* Practice
* Video

[Chinese Postman Problem](https://en.wikipedia.org/wiki/Route_inspection_problem) is a variation of [Eulerian circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/) problem for undirected graphs. An Euler Circuit is a closed walk that covers every edge once starting and ending position is same. Chinese Postman problem is defined for connected and undirected graph. The problem is to find shortest path or circuity that visits every edge of the graph at least once.

**If input graph contains Euler Circuit, then a solution of the problem is Euler Circuit**

An undirected and connected graph has Eulerian cycle if “all vertices have even degree“.

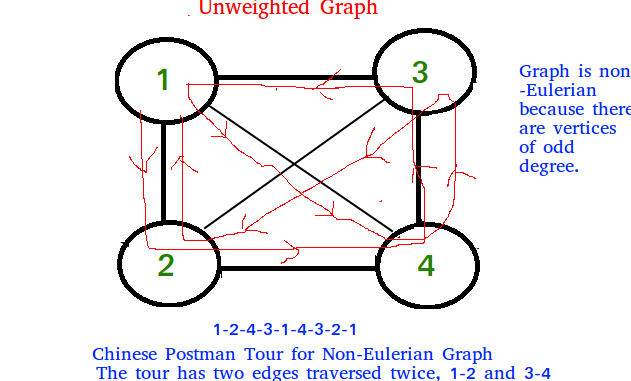


It doesn’t matter whether graph is weighted or unweighted, the Chinese Postman Route is always same as Eulerian Circuit if it exists. In weighted graph the minimum possible weight of Postman tour is sum of all edge weights which we get through Eulerian Circuit. We can’t get a shorter route as we must visit all edges at-least once.

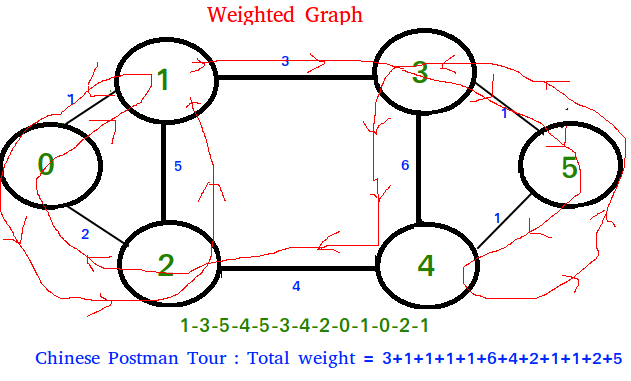
**If input graph does NOT contain Euler Circuit**

In this case, the task reduces to following.

**1)** In unweighted graph, minimum number of edges to duplicate so that the given graph converts to a graph with Eulerian Cycle.



**2)**In weighted graph, minimum total weight of edges to duplicate so that given graph converts to a graph with Eulerian Cycle.



Algorithm to find shortest closed path or optimal   
Chinese postman route in a weighted graph that may  
not be Eulerian.

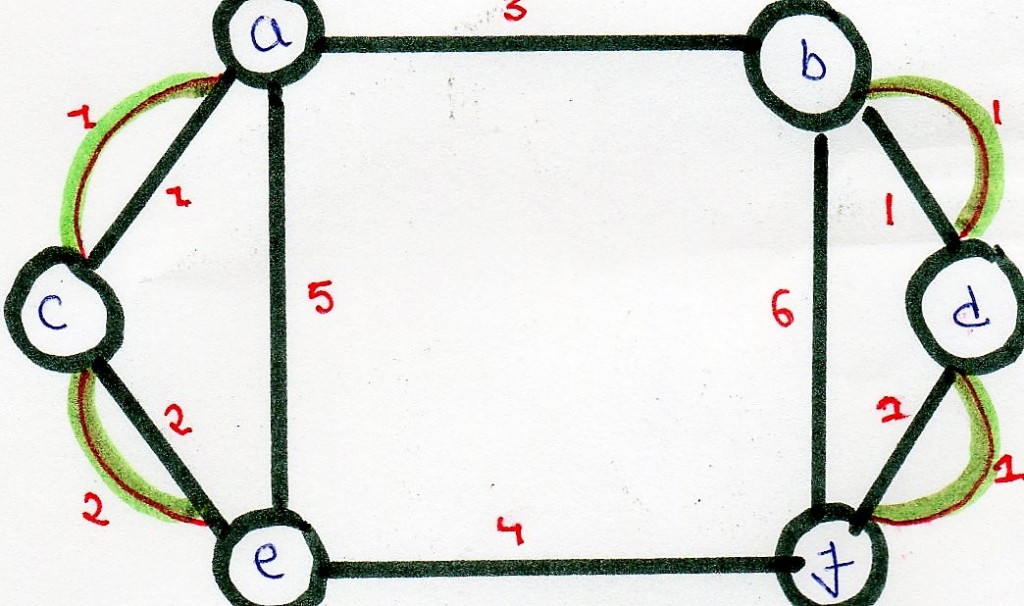
step 1 : If graph is Eulerian, return sum of all   
 edge weights.Else do following steps.  
step 2 : We find all the vertices with odd degree   
step 3 : List all possible pairings of odd vertices   
 For n odd vertices total number of pairings   
 possible are, (n-1) \* (n-3) \* (n -5)... \* 1  
step 4 : For each set of pairings, find the shortest   
 path connecting them.  
step 5 : Find the pairing with minimum shortest path   
 connecting pairs.  
step 6 : Modify the graph by adding all the edges that   
 have been found in step 5.  
step 7 : Weight of Chinese Postman Tour is sum of all   
 edges in the modified graph.  
step 8 : Print Euler Circuit of the modified graph.   
 This Euler Circuit is Chinese Postman Tour.

**Illustration :**

3  
 (a)-----------------(b)  
 1 / | | \1  
 / | | \  
 (c) | 5 6| (d)  
 \ | | /  
 2 \ | 4 | /1  
 (e)------------------(f)  
As we see above graph does not contain Eulerian circuit  
because is has odd degree vertices [a, b, e, f]  
they all are odd degree vertices .

First we make all possible pairs of odd degree vertices  
[ae, bf], [ab, ef], [af, eb]   
so pairs with min sum of weight are [ae, bf] :  
ae = (ac + ce = 3 ), bf = ( bd + df = 2 )   
Total : 5

We add edges ac, ce, bd and df to the original graph and  
create a modified graph.



Optimal chinese postman route is of length : 5 + 23 =   
28 [ 23 = sum of all edges of modified graph ]

Chinese Postman Route :   
a - b - d - f - d - b - f - e - c - a - c - e - a   
This route is Euler Circuit of the modified graph.

*From <*[*https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/*](https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/)*>*

**Hierholzer’s Algorithm for directed graph**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 27 Dec, 2022
* Read
* Discuss
* Courses
* Practice
* Video

Given a directed Eulerian graph, print an [Euler circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/). Euler circuit is a path that traverses every edge of a graph, and the path ends on the starting vertex. **Examples:**

Input : Adjacency list for the below graph

Output : 0 -> 1 -> 2 -> 0

Input : Adjacency list for the below graph

Output : 0 -> 6 -> 4 -> 5 -> 0 -> 1   
 -> 2 -> 3 -> 4 -> 2 -> 0   
Explanation:  
In both the cases, we can trace the Euler circuit   
by following the edges as indicated in the output.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We have discussed the [problem of finding out whether a given graph is Eulerian or not](https://www.geeksforgeeks.org/eulerian-path-and-circuit/). In this post, an algorithm to print the Eulerian trail or circuit is discussed. The same problem can be solved using [Fleury’s Algorithm](https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/), however, its complexity is O(E\*E). Using Hierholzer’s Algorithm, we can find the circuit/path in O(E), i.e., linear time. Below is the Algorithm: ref ( [wiki](https://en.wikipedia.org/wiki/Eulerian_path#Hierholzer.27s_algorithm) ). Remember that a directed graph has a Eulerian cycle if the following conditions are true (1) All vertices with nonzero degrees belong to a single strongly connected component. (2) In degree and out-degree of every vertex is the same. The algorithm assumes that the given graph has a Eulerian Circuit.

* Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because indegree and outdegree of every vertex must be same, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
* As long as there exists a vertex u that belongs to the current tour, but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

Thus the idea is to keep following unused edges and removing them until we get stuck. Once we get stuck, we backtrack to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used. We can use another container to maintain the final path. Let’s take an example:

Let the initial directed graph be as below

Let's start our path from 0.  
Thus, curr\_path = {0} and circuit = {}  
Now let's use the edge 0->1

Now, curr\_path = {0,1} and circuit = {}  
similarly we reach up to 2 and then to 0 again as

Now, curr\_path = {0,1,2} and circuit = {}  
Then we go to 0, now since 0 haven't got any unused  
edge we put 0 in circuit and back track till we find  
an edge

We then have curr\_path = {0,1,2} and circuit = {0}  
Similarly, when we backtrack to 2, we don't find any   
unused edge. Hence put 2 in the circuit and backtrack   
again.

curr\_path = {0,1} and circuit = {0,2}

After reaching 1 we go to through unused edge 1->3 and   
then 3->4, 4->1 until all edges have been traversed.

The contents of the two containers look as:  
curr\_path = {0,1,3,4,1} and circuit = {0,2}

now as all edges have been used, the curr\_path is   
popped one by one into the circuit.  
Finally, we've circuit = {0,2,1,4,3,1,0}

We print the circuit in reverse to obtain the path followed.  
i.e., **0->1->3->4->1->1->2->0**

Below is the implementation for the above approach:

* C++
* Java
* Python3
* C#

# Python3 program to print Eulerian circuit in given

# directed graph using Hierholzer algorithm

**def** printCircuit(adj):

    # adj represents the adjacency list of

    # the directed graph

    # edge\_count represents the number of edges

    # emerging from a vertex

    edge\_count **=** dict()

**for** i **in** range(len(adj)):

        # find the count of edges to keep track

        # of unused edges

        edge\_count[i] **=** len(adj[i])

**if** len(adj) **==** 0:

**return** # empty graph

    # Maintain a stack to keep vertices

    curr\_path **=** []

    # vector to store final circuit

    circuit **=** []

    # start from any vertex

    curr\_path.append(0)

    curr\_v **=** 0 # Current vertex

**while** len(curr\_path):

        # If there's remaining edge

**if** edge\_count[curr\_v]:

            # Push the vertex

            curr\_path.append(curr\_v)

            # Find the next vertex using an edge

            next\_v **=** adj[curr\_v][**-**1]

            # and remove that edge

            edge\_count[curr\_v] **-=** 1

            adj[curr\_v].pop()

            # Move to next vertex

            curr\_v **=** next\_v

        # back-track to find remaining circuit

**else**:

            circuit.append(curr\_v)

            # Back-tracking

            curr\_v **=** curr\_path[**-**1]

            curr\_path.pop()

    # we've got the circuit, now print it in reverse

**for** i **in** range(len(circuit) **-** 1, **-**1, **-**1):

**print**(circuit[i], end **=** "")

**if** i:

**print**(" -> ", end **=** "")

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Input Graph 1

    adj1 **=** [0] **\*** 3

**for** i **in** range(3):

        adj1[i] **=** []

    # Build the edges

    adj1[0].append(1)

    adj1[1].append(2)

    adj1[2].append(0)

    printCircuit(adj1)

    print()

    # Input Graph 2

    adj2 **=** [0] **\*** 7

**for** i **in** range(7):

        adj2[i] **=** []

    adj2[0].append(1)

    adj2[0].append(6)

    adj2[1].append(2)

    adj2[2].append(0)

    adj2[2].append(3)

    adj2[3].append(4)

    adj2[4].append(2)

    adj2[4].append(5)

    adj2[5].append(0)

    adj2[6].append(4)

    printCircuit(adj2)

    print()

# This code is contributed by

# sanjeev2552

**Output:**

0 -> 1 -> 2 -> 0  
0 -> 6 -> 4 -> 5 -> 0 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0

**Alternate Implementation:**Below are the improvements made from the above code

The above code kept a count of the number of edges for every vertex. This is unnecessary since we are already maintaining the adjacency list. We simply deleted the creation of edge\_count array. In the algorithm we replaced `if edge\_count[current\_v]` with `if adj[current\_v]`

The above code pushes the initial node twice to the stack. Though the way he coded the result is correct, this approach is confusing and inefficient. We eliminated this by appending the next vertex to the stack, instead of the current one.

In the main part, where the author tests the algorithm, the initialization of the adjacency lists `adj1` and `adj2`were a little weird. That potion is also improved.

* C++
* Python3

# Python3 program to print Eulerian circuit in given

# directed graph using Hierholzer algorithm

**def** printCircuit(adj):

    # adj represents the adjacency list of

    # the directed graph

**if** len(adj) **==** 0:

**return** # empty graph

    # Maintain a stack to keep vertices

    # We can start from any vertex, here we start with 0

    curr\_path **=** [0]

    # list to store final circuit

    circuit **=** []

**while** curr\_path:

        curr\_v **=** curr\_path[**-**1]

        # If there's remaining edge in adjacency list

        # of the current vertex

**if** adj[curr\_v]:

            # Find and remove the next vertex that is

            # adjacent to the current vertex

            next\_v **=** adj[curr\_v].pop()

            # Push the new vertex to the stack

            curr\_path.append(next\_v)

        # back-track to find remaining circuit

**else**:

            # Remove the current vertex and

            # put it in the circuit

            circuit.append(curr\_path.pop())

    # we've got the circuit, now print it in reverse

**for** i **in** range(len(circuit) **-** 1, **-**1, **-**1):

        print(circuit[i], end **=** "")

**if** i:

            print(" -> ", end **=** "")

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Input Graph 1

    adj1 **=** [[] **for** \_ **in** range(3)]

    # Build the edges

    adj1[0].append(1)

    adj1[1].append(2)

    adj1[2].append(0)

    printCircuit(adj1)

    print()

    # Input Graph 2

    adj2 **=** [[] **for** \_ **in** range(7)]

    adj2[0].append(1)

    adj2[0].append(6)

    adj2[1].append(2)

    adj2[2].append(0)

    adj2[2].append(3)

    adj2[3].append(4)

    adj2[4].append(2)

    adj2[4].append(5)

    adj2[5].append(0)

    adj2[6].append(4)

    printCircuit(adj2)

    print()

**Output:**

0 -> 1 -> 2 -> 0  
0 -> 6 -> 4 -> 5 -> 0 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0

**Time Complexity :** O(V+E).

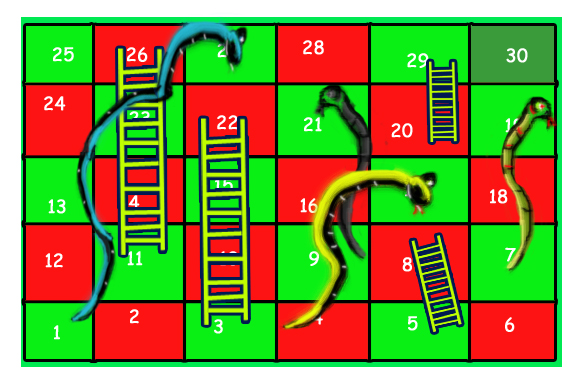
*From <*[*https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/*](https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/)*>*

**Snake and Ladder Problem**

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 19 Jul, 2022
* Read
* Discuss(150+)
* Courses
* Practice
* Video

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from the source or 1st cell. Basically, the player has total control over the outcome of the dice throw and wants to find out the minimum number of throws required to reach the last cell.

If the player reaches a cell which is the base of a ladder, the player has to climb up that ladder and if reaches a cell is the mouth of the snake, and has to go down to the tail of the snake without a dice throw.



For example, consider the board shown, the minimum number of dice throws required to reach cell 30 from cell 1 is 3.

Following are the steps:

a) First throw two dice to reach cell number 3 and then ladder to reach 22

b) Then throw 6 to reach 28.

c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

Recommended Problem

Snake and Ladder Problem

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

[DFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=DFS&sortBy=submissions)

+4 more

[Adobe](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Adobe&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+7 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/snake-and-ladder-problem4816/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 43.2K

The idea is to consider the given snake and ladder board as a directed graph with a number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if the next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from the current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find the shortest path using Breadth-First [Search](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) of the graph.

Following is the implementation of the above idea. The input is represented by two things, the first is ‘N’ which is a number of cells in the given board, second is an array ‘move[0…N-1]’ of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find minimum number

# of dice throws required to reach last

# cell from first cell of a given

# snake and ladder board

# An entry in queue used in BFS

**class** QueueEntry(object):

**def** \_\_init\_\_(self, v**=**0, dist**=**0):

        self.v **=** v

        self.dist **=** dist

'''This function returns minimum number of

dice throws required to. Reach last cell

from 0'th cell in a snake and ladder game.

move[] is an array of size N where N is

no. of cells on board. If there is no

snake or ladder from cell i, then move[i]

is -1. Otherwise move[i] contains cell to

which snake or ladder at i takes to.'''

**def** getMinDiceThrows(move, N):

    # The graph has N vertices. Mark all

    # the vertices as not visited

    visited **=** [False] **\*** N

    # Create a queue for BFS

    queue **=** []

    # Mark the node 0 as visited and enqueue it

    visited[0] **=** True

    # Distance of 0't vertex is also 0

    # Enqueue 0'th vertex

    queue.append(QueueEntry(0, 0))

    # Do a BFS starting from vertex at index 0

    qe **=** QueueEntry()  # A queue entry (qe)

**while** queue:

        qe **=** queue.pop(0)

        v **=** qe.v  # Vertex no. of queue entry

        # If front vertex is the destination

        # vertex, we are done

**if** v **==** N **-** 1:

**break**

        # Otherwise dequeue the front vertex

        # and enqueue its adjacent vertices

        # (or cell numbers reachable through

        # a dice throw)

        j **=** v **+** 1

**while** j <**=** v **+** 6 **and** j < N:

            # If this cell is already visited,

            # then ignore

**if** visited[j] **is** False:

                # Otherwise calculate its

                # distance and mark it

                # as visited

                a **=** QueueEntry()

                a.dist **=** qe.dist **+** 1

                visited[j] **=** True

                # Check if there a snake or ladder

                # at 'j' then tail of snake or top

                # of ladder become the adjacent of 'i'

                a.v **=** move[j] **if** move[j] !**= -**1 **else** j

                queue.append(a)

            j **+=** 1

    # We reach here when 'qe' has last vertex

    # return the distance of vertex in 'qe

**return** qe.dist

# driver code

N **=** 30

moves **=** [**-**1] **\*** N

# Ladders

moves[2] **=** 21

moves[4] **=** 7

moves[10] **=** 25

moves[19] **=** 28

# Snakes

moves[26] **=** 0

moves[20] **=** 8

moves[16] **=** 3

moves[18] **=** 6

**print**("Min Dice throws required is {0}".

      format(getMinDiceThrows(moves, N)))

# This code is contributed by Ajitesh Pathak

**Output**

Min Dice throws required is 3

**The time complexity** of the above solution is O(N) as every cell is added and removed only once from the queue. And a typical enqueue or dequeue operation takes O(1) time.

*From <*[*https://www.geeksforgeeks.org/snake-ladder-problem-2/*](https://www.geeksforgeeks.org/snake-ladder-problem-2/)*>*

**Minimum time required to rot all oranges**

Given a matrix of dimension M \* N where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

* 0: Empty cell
* 1: Cells have fresh oranges
* 2: Cells have rotten oranges

Determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index (i,j ) can rot other fresh oranges which are its neighbours (up, down, left and right). If it is impossible to rot every orange then simply return -1.

**Examples:**

***Input:****arr[][C] = { {2, 1, 0, 2, 1}, {1, 0, 1, 2, 1}, {1, 0, 0, 2, 1}};*

***Output:****2*

***Explanation:****At 0th time frame:*

*{2, 1, 0, 2, 1}*

*{1, 0, 1, 2, 1}*

*{1, 0, 0, 2, 1}*

*At 1st time frame:*

*{2, 2, 0, 2, 2}*

*{2, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*At 2nd time frame:*

*{2, 2, 0, 2, 2}*

*{2, 0, 2, 2, 2}*

*{2, 0, 0, 2, 2}*

***Input:****arr[][C] = { {2, 1, 0, 2, 1}, {0, 0, 1, 2, 1}, {1, 0, 0, 2, 1}}*

***Output:****-1*

***Explanation:****At 0th time frame:*

*{2, 1, 0, 2, 1}*

*{0, 0, 1, 2, 1}*

*{1, 0, 0, 2, 1}*

*At 1st time frame:*

*{2, 2, 0, 2, 2}*

*{0, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*At 2nd time frame:*

*{2, 2, 0, 2, 2}*

*{0, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*The 1 at the bottom left corner of the matrix is never rotten.*

Recommended Problem

Rotten Oranges

[Graph](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Graph&sortBy=submissions)

[Matrix](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Matrix&sortBy=submissions)

+2 more

[Accolite](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Accolite&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+5 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/rotten-oranges2536/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 83.1K

**Naive Approach:**

*The idea is very basic. Traverse through all oranges in multiple rounds. In every round, rot the oranges to the adjacent position of oranges that were rotten in the last round.*

Follow the steps below to solve the problem:

* Create a variable **no = 2** and **changed = false**.
* Run a loop until there is no cell of the matrix which is changed in an iteration.
* Run a nested loop and traverse the matrix:
* If the element of the matrix is equal to **no** then assign the adjacent elements to **no + 1** if the adjacent element’s value is equal to 1, i.e. not rotten, and update **changed** to true.
* Traverse the matrix and check if there is any cell that is **1**.
* If 1 is present return -1
* Else return **no – 2**.

Below is the implementation of the above approach.

* C++14
* Java
* Python3
* C#
* Javascript

# Python3 program to rot all

# oranges when u can move

# in all the four direction

# from a rotten orange

R **=** 3

C **=** 5

# Check if i, j is under the

# array limits of row and

# column

**def** issafe(i, j):

**if** (i >**=** 0 **and** i < R **and**

            j >**=** 0 **and** j < C):

**return** True

**return** False

**def** rotOranges(v):

    changed **=** False

    no **=** 2

**while** (True):

**for** i **in** range(R):

**for** j **in** range(C):

                # Rot all other oranges

                # present at (i+1, j),

                # (i, j-1), (i, j+1),

                # (i-1, j)

**if** (v[i][j] **==** no):

**if** (issafe(i **+** 1, j) **and**

                            v[i **+** 1][j] **==** 1):

                        v[i **+** 1][j] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i, j **+** 1) **and**

                            v[i][j **+** 1] **==** 1):

                        v[i][j **+** 1] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i **-** 1, j) **and**

                            v[i **-** 1][j] **==** 1):

                        v[i **-** 1][j] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i, j **-** 1) **and**

                            v[i][j **-** 1] **==** 1):

                        v[i][j **-** 1] **=** v[i][j] **+** 1

                        changed **=** True

        # if no rotten orange found

        # it means all oranges rottened

        # now

**if** (**not** changed):

**break**

        changed **=** False

        no **+=** 1

**for** i **in** range(R):

**for** j **in** range(C):

            # if any orange is found

            # to be not rotten then

            # ans is not possible

**if** (v[i][j] **==** 1):

**return -**1

    # Because initial value

    # for a rotten orange was 2

**return** no **-** 2

# Driver function

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    v **=** [[2, 1, 0, 2, 1],

         [1, 0, 1, 2, 1],

         [1, 0, 0, 2, 1]]

    print("Max time incurred: ",

          rotOranges(v))

# This code is contributed by Chitranayal

**Output**

Max time incurred: 2

**Time Complexity**: O((R\*C) \* (R \*C)),

* The matrix needs to be traversed again and again until there is no change in the matrix, that can happen max(R \*C)/2 times.
* So time complexity is O((R \* C) \* (R \*C)).

**Auxiliary Space:**O(1), No extra space is required.

**Minimum time required to rot all oranges using**[Breadth First Search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)**:**

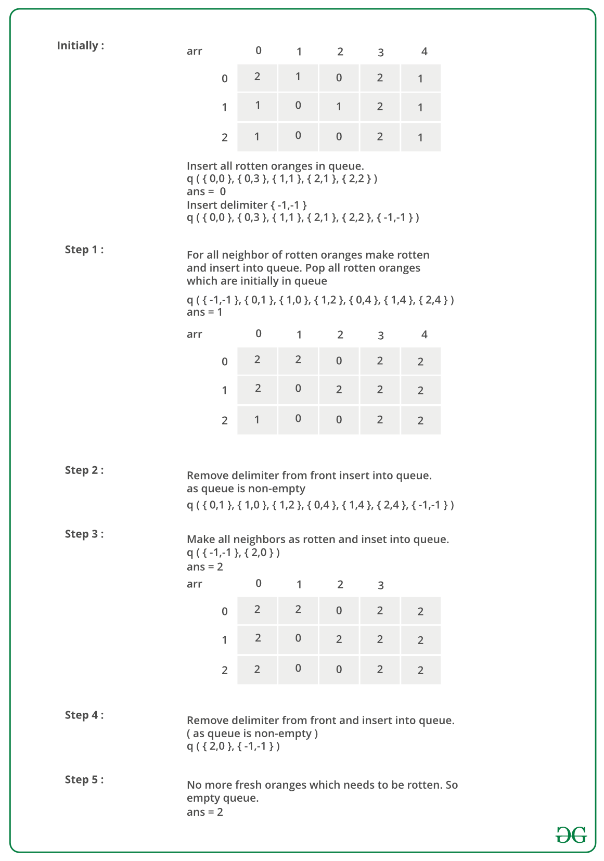
*The idea is to use*[*Breadth First Search*](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)*. The condition of oranges getting rotten is when they come in contact with other rotten oranges. This is similar to a breadth-first search where the graph is divided into layers or circles and the search is done from lower or closer layers to deeper or higher layers.*

*In the previous approach, the idea was based on BFS but the implementation was poor and inefficient. To find the elements whose values are****no****the whole matrix had to be traversed. So time can be reduced by using this efficient approach of BFS.*

Follow the steps below to solve the problem:

* Create an empty queue **Q**.
* Find all rotten oranges and enqueue them to **Q**. Also, enqueue a delimiter to indicate the beginning of the next time frame.
* Run a loop While Q is not empty and do the following while the delimiter in **Q** is not reached
* Dequeue an orange from the queue, and rot all adjacent oranges.
* While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.
* Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.
* Return the last time frame.

**Illustration:**



Below is the implementation of the above approach.

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find minimum time required to make all

# oranges rotten

**from** collections **import** deque

# function to check whether a cell is valid / invalid

**def** isvalid(i, j):

**return** (i >**=** 0 **and** j >**=** 0 **and** i < 3 **and** j < 5)

# Function to check whether the cell is delimiter

# which is (-1, -1)

**def** isdelim(temp):

**return** (temp[0] **== -**1 **and** temp[1] **== -**1)

# Function to check whether there is still a fresh

# orange remaining

**def** checkall(arr):

**for** i **in** range(3):

**for** j **in** range(5):

**if** (arr[i][j] **==** 1):

**return** True

**return** False

# This function finds if it is

# possible to rot all oranges or not.

# If possible, then it returns

# minimum time required to rot all,

# otherwise returns -1

**def** rotOranges(arr):

    # Create a queue of cells

    Q **=** deque()

    temp **=** [0, 0]

    ans **=** 1

    # Store all the cells having

    # rotten orange in first time frame

**for** i **in** range(3):

**for** j **in** range(5):

**if** (arr[i][j] **==** 2):

                temp[0] **=** i

                temp[1] **=** j

                Q.append([i, j])

    # Separate these rotten oranges

    # from the oranges which will rotten

    # due the oranges in first time

    # frame using delimiter which is (-1, -1)

    temp[0] **= -**1

    temp[1] **= -**1

    Q.append([**-**1, **-**1])

    # print(Q)

    # Process the grid while there are

    # rotten oranges in the Queue

**while** False:

        # This flag is used to determine

        # whether even a single fresh

        # orange gets rotten due to rotten

        # oranges in current time

        # frame so we can increase

        # the count of the required time.

        flag **=** False

**print**(len(Q))

        # Process all the rotten

        # oranges in current time frame.

**while not** isdelim(Q[0]):

            temp **=** Q[0]

**print**(len(Q))

            # Check right adjacent cell that if it can be rotten

**if** (isvalid(temp[0] **+** 1, temp[1]) **and** arr[temp[0] **+** 1][temp[1]] **==** 1):

                # if this is the first orange to get rotten, increase

                # count and set the flag.

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                # Make the orange rotten

                arr[temp[0] **+** 1][temp[1]] **=** 2

                # append the adjacent orange to Queue

                temp[0] **+=** 1

                Q.append(temp)

                temp[0] **-=** 1  # Move back to current cell

            # Check left adjacent cell that if it can be rotten

**if** (isvalid(temp[0] **-** 1, temp[1]) **and** arr[temp[0] **-** 1][temp[1]] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0] **-** 1][temp[1]] **=** 2

                temp[0] **-=** 1

                Q.append(temp)  # append this cell to Queue

                temp[0] **+=** 1

            # Check top adjacent cell that if it can be rotten

**if** (isvalid(temp[0], temp[1] **+** 1) **and** arr[temp[0]][temp[1] **+** 1] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0]][temp[1] **+** 1] **=** 2

                temp[1] **+=** 1

                Q.append(temp)  # Push this cell to Queue

                temp[1] **-=** 1

            # Check bottom adjacent cell if it can be rotten

**if** (isvalid(temp[0], temp[1] **-** 1) **and** arr[temp[0]][temp[1] **-** 1] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0]][temp[1] **-** 1] **=** 2

                temp[1] **-=** 1

                Q.append(temp)  # append this cell to Queue

            Q.popleft()

        # Pop the delimiter

        Q.popleft()

        # If oranges were rotten in

        # current frame than separate the

        # rotten oranges using delimiter

        # for the next frame for processing.

**if** (len(Q) **==** 0):

            temp[0] **= -**1

            temp[1] **= -**1

            Q.append(temp)

        # If Queue was empty than no rotten oranges left to process so exit

    # Return -1 if all arranges could not rot, otherwise return ans.

**return** ans **+** 1 **if**(checkall(arr)) **else -**1

# Driver program

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [[2, 1, 0, 2, 1],

           [1, 0, 1, 2, 1],

           [1, 0, 0, 2, 1]]

    ans **=** rotOranges(arr)

**if** (ans **== -**1):

**print**("All oranges cannot rotn")

**else**:

**print**("Time required for all oranges to rot => ", ans)

        # This code is contributed by mohit kumar 29

**Output**

Time required for all oranges to rot => 2

**Time Complexity:** O( R \*C), Each element of the matrix can be inserted into the queue only once so the upper bound of iteration is O(R\*C)

**Auxiliary Space:** O(R\*C), To store the elements in a queue

*From <*[*https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/*](https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/)*>*

**Determine whether a universal sink exists in a directed graph**

Determine whether a universal sink exists in a directed graph. A universal sink is a vertex which has no edge emanating from it, and all other vertices have an edge towards the sink.

Input :   
v1 -> v2 (implies vertex 1 is connected to vertex 2)  
v3 -> v2  
v4 -> v2  
v5 -> v2  
v6 -> v2   
Output :  
Sink found at vertex 2

Input :   
v1 -> v6  
v2 -> v3  
v2 -> v4  
v4 -> v3  
v5 -> v3  
Output :  
No Sink

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We try to eliminate n – 1 non-sink vertices in **O(n)** time and check the remaining vertex for the sink property.

To eliminate vertices, we check whether a particular index (A[i][j]) in the adjacency matrix is a 1 or a 0. If it is a 0, it means that the vertex corresponding to index j cannot be a sink. If the index is a 1, it means the vertex corresponding to i cannot be a sink. We keep increasing i and j in this fashion until either i or j exceeds the number of vertices.

Using this method allows us to carry out the universal sink test for only one vertex instead of all n vertices. Suppose we are left with only vertex i.

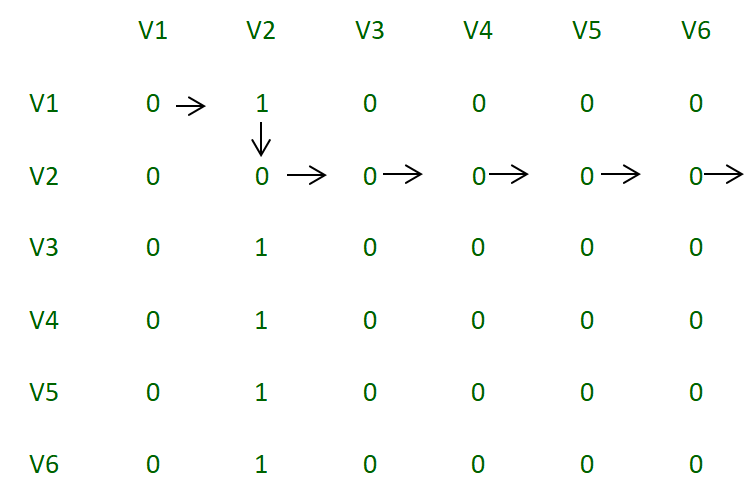
We now check for whether row i has only 0s and whether row j as only 1s except for A[i][i], which will be 0.

**Illustration :**

v1 -> v2   
v3 -> v2  
v4 -> v2  
v5 -> v2  
v6 -> v2   
We can visualize the adjacency matrix for   
the above as follows:  
0 1 0 0 0 0  
0 0 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0

We observe that vertex 2 does not have any emanating edge, and that every other vertex has an edge in vertex 2. At A[0][0] (A[i][j]), we encounter a 0, so we increment j and next look at A[0][1]. Here we encounter a 1. So we have to increment i by 1. A[1][1] is 0, so we keep increasing j.

We notice that A[1][2], A[1][3].. etc are all 0, so j will exceed the number of vertices (6 in this example). We now check row i and column i for the sink property. Row i must be completely 0, and column i must be completely 1 except for the index A[i][i]

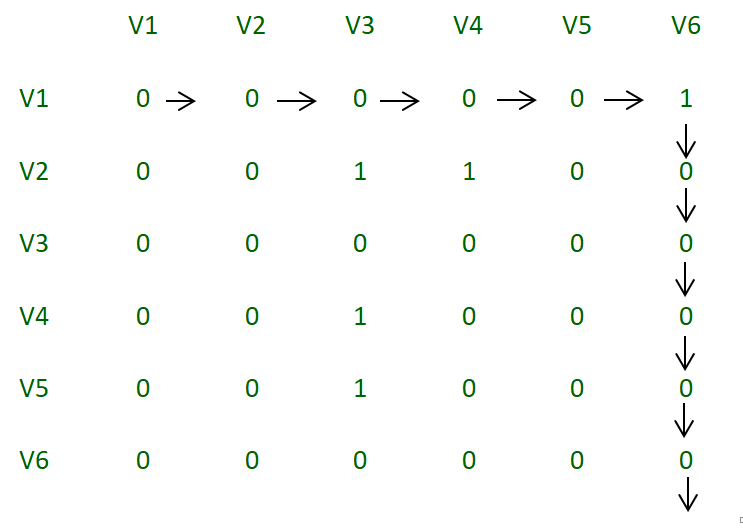
[](https://media.geeksforgeeks.org/wp-content/uploads/Adjacency_matrix_2.png)

*Adjacency Matrix*

**Second Example:**

v1 -> v6  
v2 -> v3  
v2 -> v4  
v4 -> v3  
v5 -> v3  
We can visualize the adjacency matrix  
for the above as follows:  
0 0 0 0 0 1  
0 0 1 1 0 0  
0 0 0 0 0 0  
0 0 1 0 0 0  
0 0 1 0 0 0  
0 0 0 0 0 0

In this example, we observer that in row 1, every element is 0 except for the last column. So we will increment j until we reach the 1. When we reach 1, we increment i as long as the value of A[i][j] is 0. If i exceeds the number of vertices, it is not possible to have a sink, and in this case, i will exceed the number of vertices.

[](https://media.geeksforgeeks.org/wp-content/uploads/Adjacency_matrix.png)

*Adjacency Matrix*

**Implementation:**

* Java
* Python3
* C#
* Javascript

# Python3 program to find whether a

# universal sink exists in a directed graph

**class** Graph:

    # constructor to initialize number of

    # vertices and size of adjacency matrix

**def** \_\_init\_\_(self, vertices):

        self.vertices **=** vertices

        self.adjacency\_matrix **=** [[0 **for** i **in** range(vertices)]

**for** j **in** range(vertices)]

**def** insert(self, s, destination):

        # make adjacency\_matrix[i][j] = 1

        # if there is an edge from i to j

        self.adjacency\_matrix[s **-** 1][destination **-** 1] **=** 1

**def** issink(self, i):

**for** j **in** range(self.vertices):

            # if any element in the row i is 1, it means

            # that there is an edge emanating from the

            # vertex, which means it cannot be a sink

**if** self.adjacency\_matrix[i][j] **==** 1:

**return** False

            # if any element other than i in the column

            # i is 0, it means that there is no edge from

            # that vertex to the vertex we are testing

            # and hence it cannot be a sink

**if** self.adjacency\_matrix[j][i] **==** 0 **and** j !**=** i:

**return** False

        # if none of the checks fails, return true

**return** True

    # we will eliminate n-1 non sink vertices so that

    # we have to check for only one vertex instead of

    # all n vertices

**def** eliminate(self):

        i **=** 0

        j **=** 0

**while** i < self.vertices **and** j < self.vertices:

            # If the index is 1, increment the row

            # we are checking by 1

            # else increment the column

**if** self.adjacency\_matrix[i][j] **==** 1:

                i **+=** 1

**else**:

                j **+=** 1

        # If i exceeds the number of vertices, it

        # means that there is no valid vertex in

        # the given vertices that can be a sink

**if** i > self.vertices:

**return -**1

**else if** self.issink(i) **is** False:

**return -**1

**else**:

**return** i

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    number\_of\_vertices **=** 6

    number\_of\_edges **=** 5

    g **=** Graph(number\_of\_vertices)

    # input set 1

    # g.insert(1, 6)

    # g.insert(2, 6)

    # g.insert(3, 6)

    # g.insert(4, 6)

    # g.insert(5, 6)

    # input set 2

    g.insert(1, 6)

    g.insert(2, 3)

    g.insert(2, 4)

    g.insert(4, 3)

    g.insert(5, 3)

    vertex **=** g.eliminate()

    # returns 0 based indexing of vertex.

    # returns -1 if no sink exits.

    # returns the vertex number-1 if sink is found

**if** vertex >**=** 0:

**print**("Sink found at vertex %d" **%** (vertex **+** 1))

**else**:

**print**("No Sink")

# This code is contributed by

# sanjeev2552

**Output**

No Sink

This program eliminates non-sink vertices in **O(n)** complexity and checks for the sink property in **O(n)** complexity.

*From <*[*https://www.geeksforgeeks.org/determine-whether-universal-sink-exists-directed-graph/*](https://www.geeksforgeeks.org/determine-whether-universal-sink-exists-directed-graph/)*>*

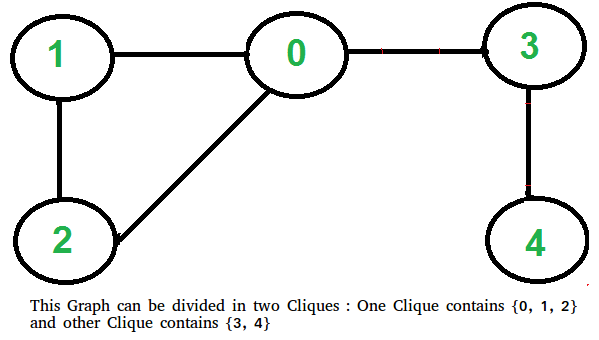
**Two Clique Problem (Check if Graph can be divided in two Cliques)**

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 11 Jun, 2022
* Read
* Discuss
* Courses
* Practice
* Video

A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other. Given a Graph, find if it can be divided into two Cliques.

**Examples:**

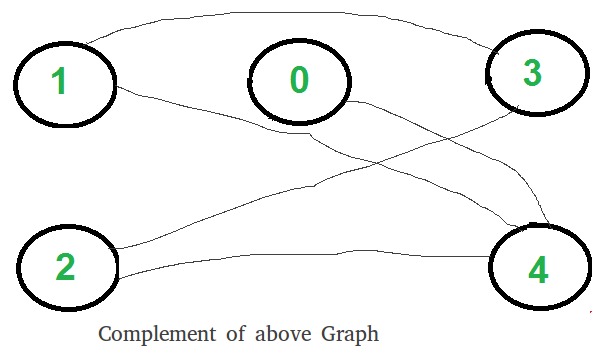
Input : G[][] = {{0, 1, 1, 0, 0},  
 {1, 0, 1, 1, 0},  
 {1, 1, 0, 0, 0},  
 {0, 1, 0, 0, 1},  
 {0, 0, 0, 1, 0}};  
Output : Yes



[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

This problem looks tricky at first, but has a simple and interesting solution. A graph can be divided in two cliques if its complement graph is [Bipartitite](https://www.geeksforgeeks.org/bipartite-graph/). So below are two steps to find if graph can be divided in two Cliques or not.

* Find the complement of Graph. Below is the complement graph is above shown graph. In complement, all original edges are removed. And the vertices which did not have an edge between them, now have an edge connecting them.



* Return true if complement is Bipartite, else false. The above shown graph is Bipartite. Checking whether a Graph is Bipartite or no is discussed [here](https://www.geeksforgeeks.org/bipartite-graph/).

**How does this work?**

If complement is Bipartite, then graph can be divided into two sets U and V such that there is no edge connecting to vertices of same set. This means in original graph, these sets U and V are completely connected. Hence original graph could be divided in two Cliques.

**Implementation:**

Below is the implementation of above steps.

* C++
* Java
* Python3
* Javascript

# Python3 program to find out whether a given

# graph can be converted to two Cliques or not.

**from** queue **import** Queue

# This function returns true if subgraph

# reachable from src is Bipartite or not.

**def** isBipartiteUtil(G, src, colorArr):

**global** V

    colorArr[src] **=** 1

    # Create a queue (FIFO) of vertex numbers

    # and enqueue source vertex for BFS traversal

    q **=** Queue()

    q.put(src)

    # Run while there are vertices in

    # queue (Similar to BFS)

**while** (**not** q.empty()):

        # Dequeue a vertex from queue

        u **=** q.get()

        # Find all non-colored adjacent vertices

**for** v **in** range(V):

            # An edge from u to v exists and

            # destination v is not colored

**if** (G[u][v] **and** colorArr[v] **== -**1):

                # Assign alternate color to this

                # adjacent v of u

                colorArr[v] **=** 1 **-** colorArr[u]

                q.put(v)

            # An edge from u to v exists and destination

            # v is colored with same color as u

**elif** (G[u][v] **and** colorArr[v] **==** colorArr[u]):

**return** False

    # If we reach here, then all adjacent

    # vertices can be colored with alternate color

**return** True

# Returns true if a Graph G[][] is Bipartite or not.

# Note that G may not be connected.

**def** isBipartite(G):

**global** V

    # Create a color array to store colors assigned

    # to all vertices. Vertex number is used as index

    # in this array. The value '-1' of colorArr[i]

    # is used to indicate that no color is assigned

    # to vertex 'i'. The value 1 is used to indicate

    # first color is assigned and value 0 indicates

    # second color is assigned.

    colorArr **=** [**-**1] **\*** V

    # One by one check all not yet

    # colored vertices.

**for** i **in** range(V):

**if** (colorArr[i] **== -**1):

**if** (isBipartiteUtil(G, i, colorArr) **==** False):

**return** False

**return** True

# Returns true if G can be divided into

# two Cliques, else false.

**def** canBeDividedinTwoCliques(G):

**global** V

    # Find complement of G[][]

    # All values are complemented

    # except diagonal ones

    GC **=** [[None] **\*** V **for** i **in** range(V)]

**for** i **in** range(V):

**for** j **in** range(V):

            GC[i][j] **= not** G[i][j] **if** i !**=** j **else** 0

    # Return true if complement is

    # Bipartite else false.

**return** isBipartite(GC)

# Driver Code

V **=** 5

G **=** [[0, 1, 1, 1, 0],

     [1, 0, 1, 0, 0],

     [1, 1, 0, 0, 0],

     [0, 1, 0, 0, 1],

     [0, 0, 0, 1, 0]]

**if** canBeDividedinTwoCliques(G):

**print**("Yes")

**else**:

    print("No")

# This code is contributed by PranchalK

**Output :**

Yes

Time complexity of above implementation is O(V2).