**Algorithm Theory Assignment 01 (Answer Script)**

|  |
| --- |
| Question No. 01` |
| Write a program that takes a directed graph as input and checks whether it is bi-directional which means for every pair of nodes where there is an edge u -> v, there should also be an edge v -> u.  Which graph representation would you prefer? |
| Answer:  #include<bits/stdc++.h>  using namespace std;  int main(){  int n,m; // nodes, edges  cin>>n>>m;  int matrix[n][n]={};  int u,v;  // input  for(int i=0; i<m; i++){  cin>>u>>v;  matrix[u][v]=1;  // matrix[v][u]=1;  }  cout<<endl;  // matrix print  for(int i=0; i<n; i++){  for(int j=0; j<n; j++){  cout<<matrix[i][j]<<" ";  }  cout<<endl;  }    // check bidirectionality  int flag=1; // default: bi.  for(int i=0; i<n; i++){  for(int j=0; j<n; j++){  // cout<<matrix[i][j]<<" ";  if (matrix[i][j]==1 && matrix[j][i]!=matrix[i][j]){  flag=0; // not bi.  }  }  }  cout<<endl;  if(flag==1){  cout<<"bi-directional"<<endl;  }  else{  cout<<"not bi-directional"<<endl;  }  } |
| In this case, I prefer adjacency matrix representation over adjacency list representation, because in the matrix representation checking an existing edge between two vertices it can be done in O(1) time whereas in an adjacency list we may have to search through all the vertices adjacent to every single vertex which cause time complexity of O(V). |

|  |
| --- |
| Question No. 02 |
| This is a follow-up of problem 1. Trim all edges of the input graph where there is an edge from u -> v but not v -> u. Output the number of edges you have to remove. |
| Answer:  #include<bits/stdc++.h>  using namespace std;  int main(){  int n,m; // nodes, edges  cin>>n>>m;  int matrix[n][n]={};  int u,v;  // input  for(int i=0; i<m; i++){  cin>>u>>v;  matrix[u][v]=1;  // matrix[v][u]=1;  }  cout<<endl;  // matrix print  for(int i=0; i<n; i++){  for(int j=0; j<n; j++){  cout<<matrix[i][j]<<" ";  }  cout<<endl;  }    // check bidirectionality  int flag=1; // default: bi.  int count=0; // count non bi-dir. edges which nee to remove  for(int i=0; i<n; i++){  for(int j=0; j<n; j++){  // cout<<matrix[i][j]<<" ";  if (matrix[i][j]==1 && matrix[j][i]!=matrix[i][j]){  flag=0; // not bi.  count++;  matrix[i][j]=0;  }  }  }  cout<<"after trimming (if required)"<<endl;  // matrix print again after trimming  for(int i=0; i<n; i++){  for(int j=0; j<n; j++){  cout<<matrix[i][j]<<" ";  }  cout<<endl;  }  cout<<endl;  if(flag==1){  cout<<"bi-directional."<<endl<<"need to remove minimum "<<count<<" edges to make it bi-directional."<<endl;  }  else{  cout<<"not bi-directional."<<endl<<"need to remove minimum "<<count<<" edges to make it bi-directional."<<endl;  }  } |

|  |
| --- |
| Question No. 03 |
| Connected component in an **undirected** graph means  a set of vertices in a graph that are linked to each other by paths. For example, **10** - There are 2 connected components in the following graph https://lh3.googleusercontent.com/boKTtT8j1Ne2NS8Srbc9SfZq57FB4HAespV0_HZPXRS8ntpWp8IEFhtDppk4DpUqbduHOhFF0INbMNHn1ZKOkmHDRXzN6JPV6GK9zu-wm87PwHRnfWp1DA6WCyGDcpt6zvZhbBqPq3uisixExoQlOl39FbkLbooE4iaKjlWOGwOIZA4JfR_5WwfqZaXQGQ - There are 3 connected components in the following graph https://lh6.googleusercontent.com/Tw_SMIbsnagNZq15ipE8wFtA_1MgvYA3qa9BRUCw2grN2E4Tgg5Go0GeE7DUSbfLaQ8qc-0IHzNBO8m2ZJ_w4DRVLetbKVWrAtH_plaHhy2b-HJRRrtDUDvdVyKanhSZuBMP-E8nJ8PbXwYgYl4vYI9bHG8TH1bPT2MxpRm2bGmq4hDGo2W6Z1QNLrYaTQ Write a program to take an **undirected** graph as input and count the number of connected components in it. |
| Answer:  #include<bits/stdc++.h>  using namespace std;  const int N=1e5; //visited array size  int visited[N];  vector<int>adj\_list[N];  void DFS(int node){  if (visited[node]==0){ // print only when not visited  cout<<node<<" ";  }  visited[node]=1;  for(int adj\_node: adj\_list[node]){  if(visited[adj\_node]==0){  DFS(adj\_node);  }  }  }  int main(){  int n,m;  cin>>n>>m;  for(int i=0; i<m; i++){  int u,v;  cin>>u>>v;  adj\_list[u].push\_back(v);  adj\_list[v].push\_back(u);  }  cout<<endl;  // call DFS  int src=1;  DFS(src);  cout<<endl;  int count=1;  for(int i=0; i<n; i++){  // cout<<visited[i]<<" ";  if(visited[i]!=1){  count++;  DFS(i);  cout<<endl;  }  }  cout<<endl<<"connected components: "<<count;    // this only works when source is 0  // int src=0;  // for(int i=src; i<n; i++){  // DFS(i);  // cout<<endl;  // }  } |

|  |
| --- |
| Question No. 04 |
| Your friend was working with the following **undirected** graph and he was simulating **BFS** on it. But he accidentally selected both **node A** and **node E** as source. Which means he selected both nodes **A & E**, marked them as **visited** and pushed them into the queue initially. Then he went on with performing normal BFS. **15**  Simulate the resulting BFS for your friend. What is the resulting traversal order and what will the level of each node be. Assume that both nodes **A & E** are at level **0**. You don’t need to write code. https://lh6.googleusercontent.com/w-A6aOvtEoyl-lse96N2Bv2xZkksg_Y46yoFLFuGtQJdvaotB5ThzSfxxL4cSZwI3zv1iunL354LEOfvtYHDFtnO3WdcmDvkCr_Y-hs6FYJCr_c6HhdpjUhP1QP6Lv5X2ZUvLvon3z0cPB7x5Nbo1-aowwywE9e6wCA5y3nVFXWfVkJ7Scb3ozZx81AyVQ |
| |  |  |  | | --- | --- | --- | | Queue | Visited array | Output | | A, E | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` |  |  |  | 1 | | A | B | C | D | E | |  | | Pop A and print.  A adjacent node: B, E.  B pushed into the queue. | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 |  |  | 1 | | A | B | C | D | E | | A | | E, B | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 |  |  | 1 | | A | B | C | D | E | |  | | Pop E and print.  E adjacent node: A, B, D.  D pushed into the queue. | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 |  | 1 | 1 | | A | B | C | D | E | | A, E | |  |  |  | | B, D | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 |  | 1 | 1 | | A | B | C | D | E | |  | | Pop B and print.  B adjacent node: A, E, C.  C pushed into the queue. | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 | 1 | 1 | 1 | | A | B | C | D | E | | A, E, B | | D, C | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 | 1 | 1 | 1 | | A | B | C | D | E | |  | | Pop D and print.  D adjacent node: E, C. | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 | 1 | 1 | 1 | | A | B | C | D | E | | A, E, B, D | | C | |  |  |  |  |  | | --- | --- | --- | --- | --- | | 1` | 1 | 1 | 1 | 1 | | A | B | C | D | E | |  | | Pop C and print.  C adjacent node: B, D. |  | A, E, B, D, C |  |  |  | | --- | --- | | Level 0 | A, E.  As they are source nodes. | | Level 1 | B, D  As they are connected to the source nodes.  From the queue, we can see that B comes as an adjacent node of A and D comes as an adjacent node of E. | | Level 2 | C  it is connected to the level 1 node.  From the queue, we can see that C comes as an adjacent node of D. | |

|  |
| --- |
| Question No. 05 |
| The int\_to\_binary() function using recursion. **10** |
| Answer:  #include<bits/stdc++.h>  using namespace std;  void int\_to\_binary(int decimal){  // base case  if(decimal==0){  return;  }  else{  int\_to\_binary(decimal/2);  cout<<(decimal%2);  }  }  int main(){  int decimal;  cin>>decimal;  int\_to\_binary(decimal);  // string res= int\_to\_binary(decimal);  // cout<<res;  } |

|  |
| --- |
| Question No. 06 |
| The gcd() function using recursion. **10** |
| Answer:  #include<bits/stdc++.h>  using namespace std;  int gcd(int a, int b){ //a>b  if(b==0){  return a;  }  int rem=a%b;  a=b;  b=rem;  gcd(a,b);  }  int main(){  int a,b,res;  cin>>a>>b;  if(a>b){  res=gcd(a,b);  }  else{  res=gcd(b,a);  }  cout<<res;  } |

|  |
| --- |
| Question No. 07 |
| Calculate the time complexity of the following code snippet: **5**   **int count = 0;**  **for (int i = N; i > 0; i /= 2) {**  **for (int j = 0; j < i; j++) {**  **count += 1;**  **}**  **}** |
| Answer:  In outer loop i runs for N times, then N/2 times and N/4 times and so on. For each “i”, inner for loop runs until “j” becomes equal to “i-1”. If outer loop runs for N times, inner loop runs for N-1 times.  So time complexity is O( N\*N-1 ) which is equivalent to O(N2). |

|  |
| --- |
| Question No. 08 |
| Look at the following code snippet. Calculate the time and space complexity. **10**  https://lh6.googleusercontent.com/QHFNN0RmWcTyES0zhTOYpOO8jTYg7NmxwD9C4hH0r2wSk51pccxogx-RyiSgYCkMCZXJK31tZpDJCw3qJjHz5mPD_mNmbXgGnZy02Mclx6BO4vGrjxGpIKMi1Tpe7WsuxEJH6TzniSU7Ri4s73LRwb1PBDpceuBv1Ys_K3MlqbGB7q2s9XLbS4Fmizr2Kg |
| Answer:  **Space complexity:**  Visited array has space complexity of O(N) and each node can contain maximum of b nodes as its adjacent nodes in the adj\_list vector, target\_node variable has space complexity of O(1).  Space complexity is O(Nb) + O(1) = O(NB).  **Time complexity:**  For the DFS algorithm time complexity depends on the number of nodes and edges between them, each node will be accessed once and each edge will be accessed twice  So time complexity is O (N+2E) = O (N+E).  And the nested BFS algorithm has a time complexity of O (N+E).  So total time complexity is O((N+E)2) |

|  |
| --- |
| Question No. 09 |
| Will BFS or DFS act differently on graphs having **multi-edge** and **self-loops**? Why or why not? Explain. **5** |
| Answer:  BFS and DFS do not act differently on graphs with multi-edges and self-loops because an array called visited is used to track the visited nodes.  Let, initially all nodes are zero in the visited array, so when a node is visited for the first time, that index becomes 1 in the visited array and if that node appears in the program flow, it is ignored and the program flow goes to the next adjacent node.  So self-loop and multi-edge have no effect on BFS and DFS traversal. |

|  |
| --- |
| Question No. 10 |
| We want to perform **DFS** without recursion. Can you think of any data structure that we can use to do this? **10** Show a simulation and print the traversal order for the following graph. Use **node 1** as the source.  Compare your traversal order with the actual recursive DFS traversal order. You don’t need to write code. https://lh5.googleusercontent.com/xl5deT57EBGj5LJ6M8A0EuO6F4l4PWqMSigo86t5BydVeyitylbQadHIXIvUnmvZoJ0DINb7RO7RGZaEvt5AzJEHVgOV907-2OefAKG99HczrBXpzB7ds0xsexzjHvNUIWPQSY2bR_pgRscqV9syz2JrvriUm3HeDUTH0eowJg9gHulTpgzyRQjPBB1-JA |
| Answer:  Stack Data structure can be used to implement DFS graph traversal.  **Code output snippet:**  6 8  1 4  1 5  5 0  4 0  5 4  4 2  0 2  2 3  0 -> 5 4 2  1 -> 4 5  2 -> 4 0 3  3 -> 2  4 -> 1 0 5 2  5 -> 1 0 4  **DFS traversal steps using Stack Data structure.**   |  |  |  | | --- | --- | --- | | Stack(LIFO) | Visited array | Output | | 1 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | `1 |  |  |  |  | | 0 | 1 | 2 | 3 | 4 | 5 | |  | | Pop 1 and print.  1 node adjacent: 4, 5.  4, 5 pushed into the stack. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1 | | 4,5 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1 | | Pop 5 and print (LIFO).  5 node adjacent: 1, 4, 0.  0 pushed into the stack. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5 | |  |  |  | | 4, 0 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5 | | Pop 0 and print.  0 node adjacent: 5, 4, 2.  2 pushed into the stack. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0 | | 4, 2 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0 | | Pop 2 and print.  2 node adjacent: 0, 3.  3 pushed into the stack. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0, 2 | | 4, 3 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0, 2, 3 | | Pop 3 and print.  3 node adjacent: 2. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0, 2, 3 | | 4 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0, 2, 3 | | Pop 4 and print.  4 node adjacent: 1, 5, 0. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 5, 0, 2, 3, 4 |   **DFS traversal steps using Recursion:**   |  |  |  | | --- | --- | --- | | steps | visited | output | | 1 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | `1 |  |  |  |  | | 0 | 1 | 2 | 3 | 4 | 5 | | 1 | | 1 adjacent node:  4, 5  Paused and DFS called on 4. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | |  | `1 |  |  | 1 |  | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4 | | 4 adjacent node:  1, 0, 5, 2  Paused and DFS called on 0. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 |  |  | 1 |  | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0 | | 0 adjacent node:  5, 4, 2  Paused and DFS called on 5. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5 | | 5 adjacent node:  1, 0, 4  As all nodes are already visited return to parent node 0. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 |  |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5 | | 0 adjacent node:  5, 4, 2  Paused and DFS called on 2. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 |  | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2 | | 2 adjacent node:  4, 0, 3  Paused and DFS called on 3. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2, 3 | | 3 adjacent node:  2  As all nodes are already visited return to parent node 2 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2, 3 | | 2 adjacent node:  4, 0, 3  As all nodes are already visited return to parent node 4 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2, 3 | | 4 adjacent node:  1, 0, 5, 2  As all nodes are already visited return to parent node 1 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2, 3 | | 1 adjacent node:  4, 5  As all nodes are already visited, code execution complete. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 1 | `1 | 1 | 1 | 1 | 1 | | 0 | 1 | 2 | 3 | 4 | 5 | | 1, 4, 0, 5, 2, 3 |   **Comparison:**  DFS gives different outputs when using recursion and Stack Data stricture, because their mode of operation is different.  When adjacent node of source 1 is considered, there are two nodes 4 and 5.  Stack works as LIFO, so 5 accessed first and the program continues.  In recursion, program accessed 4 first and continues.  So, two process gives two different outputs. |

Submitted by: Subrata Saha

Email:[Subratabaec@gmail.com](mailto:Subratabaec@gmail.com)

Date: 25.12.2022

THE END