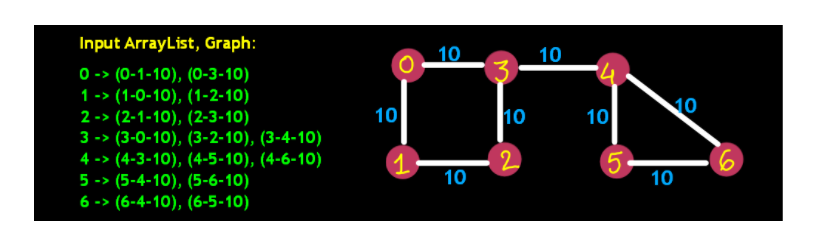
**1. Problem Discussion :**

In this problem you are given a graph, a source vertex and a destination vertex. All you have to do is find and print all paths between source and destination. Print them in lexicographical order. > Input is managed for you. Study the example below for better understanding.

****

In the above example, for the given graph, source being 0 and destination being 6, there are 4 possible paths.

**2. Approach :**

The problem here deals with finding and printing all possible paths between a given source vertex and destination vertex in a graph. To solve this problem we will use Recursion just like previous problems. Let's first take a look at hasPath function code:

ConsoleJava

public static boolean hasPath(ArrayList< Edge>[] graph, int src, int dest, boolean[] visited) {

if (src == dest) //1

return true;

visited[src] = true; //2

for (Edge e : graph[src]) { //3

if (!visited[e.nbr]) { //4

boolean nbrHasPath = hasPath(graph, e.nbr, dest, visited); //5

if (nbrHasPath) //6

return true; //7

}

}

return false; //8

}

In the hasPath() function, the return type is Boolean, which basically indicates whether there exists a path between source and destination. But in this problem, instead of checking and returning some Boolean value we need to print the existing path. And to achieve this, the first change that we make in printAllPaths() function is that we change the return type of this function to void. And in addition, we add String in the argument of the function which will store the path covered so far. Changing this return type to void also implies that printAllPaths() function returns nothing therefore it makes no sense to capture the result of its recursive calls. (Statement - 5 in above code). Which also implies the invalidation of statements 6 and 7 from above code for printAllPaths() function. Moving to the most important point; even after doing the above changes, we may get a path or few printed but what about all possible paths. To take care of this, it is really important that we explore all possibilities. In the hasPath() function we used to set the corresponding value to src in the visited array as true, so that we can stop the redundant calls. But now, to print all paths, we need to allow even the once visited vertex to contribute to other possible paths. For this, it's important to set the value, corresponding to src, in the visited array as false, after we explore all possibilities through the src vertex. Let's code this:

ConsoleJava

public static void printAllPaths(ArrayList[] graph, int src, int dest,

boolean[] visited, String psf) {

if (src == dest) {

System.out.println(psf);

return;

}

visited[src] = true;

for (Edge e : graph[src]) {

if (!visited[e.nbr]) {

printAllPaths(graph, e.nbr, dest, visited, psf + e.nbr);

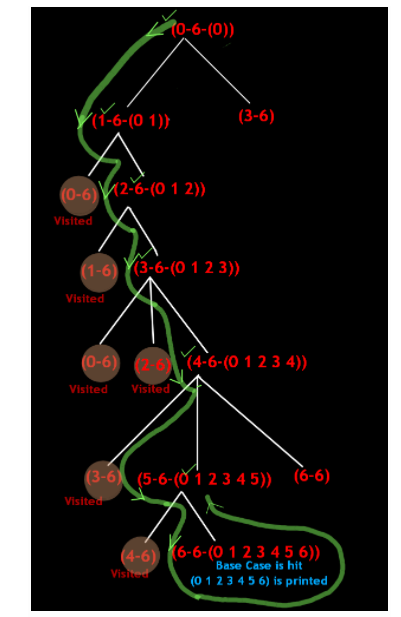
}

}

visited[src] = false;

}

Yes! That looks good. But don't feel low, if that's difficult to understand. We will understand the code even better with an example. Let's take an example and dry run this code.

****

In the above figure, you can see graph representation corresponding to the graph, Array List. Let the source be 0 and destination be 6.

ConsoleJava

public static void printAllPaths(ArrayList[] graph, int src, int dest,

boolean[] visited, String psf) {

if (src == dest) { //1

System.out.println(psf);

//1.1

return; //1.2

}

visited[src] = true; //2

for (Edge e : graph[src]) { //3

if (!visited[e.nbr]) { //3.1

printAllPaths(graph, e.nbr, dest, visited, psf + e.nbr); //3.2

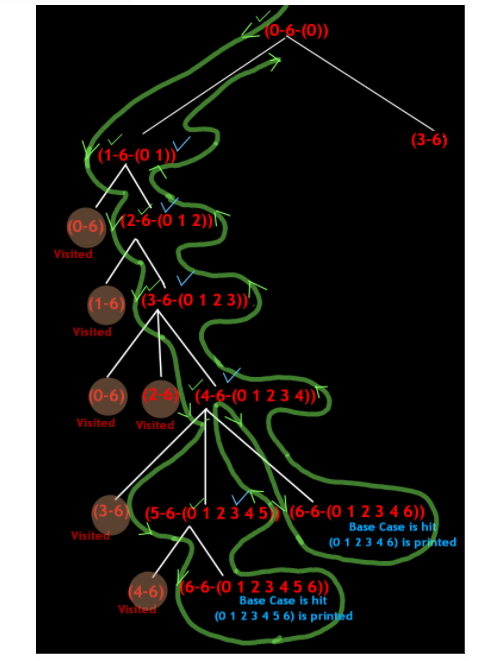
}

}

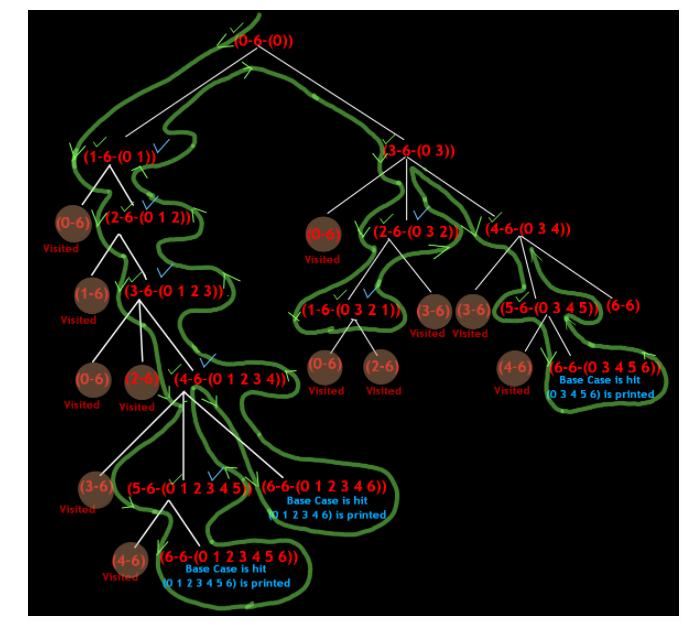
visited[src] = false; //4

}

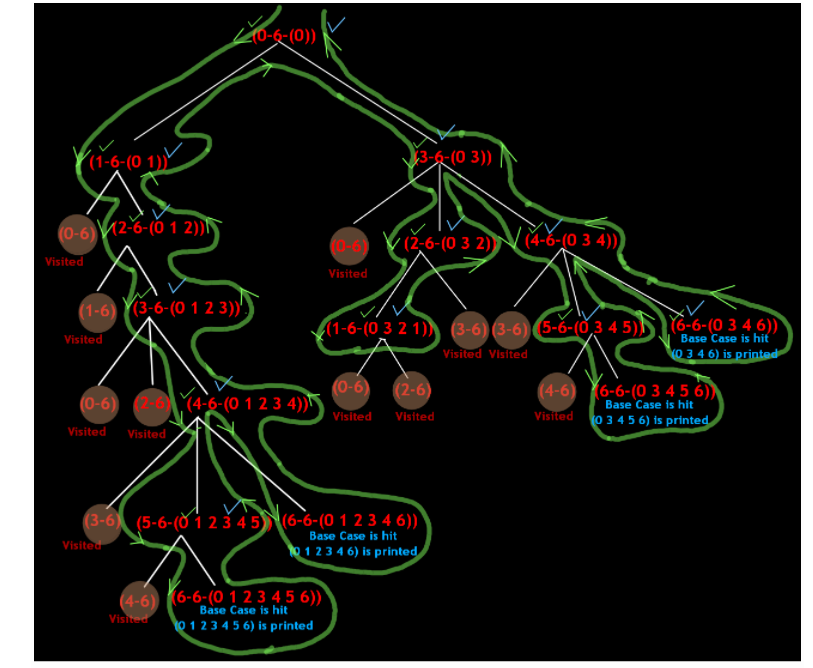
Statement-1 is false for source 0. Then control jumps to statement-2 and the value at index 0 is set true in the visited array. Then using a for loop, a recursive call to 0's neighbor is made. First neighbor is 1. Then we check whether 1 has been visited earlier or not. 1 has not been visited. Therefore, a recursive call is made to printAllPath(1, 6, "0 1"). Statement-1 is false for source 2. So control jumps to statement-2 and the value at index 2 is set true in the visited array. Then using a for loop, a recursive call to 2's neighbor is made. First neighbor is 1. Then we check whether 1 has been visited earlier or not. 1 has been visited, therefore we move to the next neighbor of 2 i.e. 3 in this case. Then we check whether 3 has been visited earlier or not. 3 has not been visited, thereforeprintAllPath(3, 6, "0 1 2 3") will be called. Statement-1 is false for source 3. So control jumps to statement-2 and the value at index 3 is set true in the visited array. Then using a for loop, a recursive call to 3's neighbor is made. First neighbor is 0. Then we check whether 0 has been visited earlier or not. 0 has been visited, therefore we move to the next neighbor of 3 i.e. 2 in this case. Then we check whether 2 has been visited earlier or not. 2 has been visited, therefore we move to the next neighbor of 3 i.e. 4 in this case. Then we check whether 4 has been visited earlier or not. 4 hasn't been visited, therefore printAllPath(4, 6, "0 1 2 3 4") will be called. Statement-1 is false for source 4. So control jumps to statement-2 and the value at index 4 is set true in the visited array. Then using a for loop, a recursive call to 4's neighbor is made. First neighbor is 3. Then we check whether 3 has been visited earlier or not. 3 has been visited, therefore we move to the next neighbor of 4 i.e. 5 in this case. Then we check whether 5 has been visited earlier or not. 5 hasn't been visited, therefore printAllPath(5, 6, "0 1 2 3 4 5") will be called. Statement-1 is false for source 5. So control jumps to statement-2 and the value at index 5 is set true in the visited array. Then using a for loop, a recursive call to 5's neighbor is made. First neighbor is 4. Then we check whether 4 has been visited earlier or not. 4 has been visited, therefore we move to the next neighbor of 5 i.e. 6 in this case. Then we check whether 6 has been visited earlier or not. 6 hasn't been visited, therefore printAllPath(6, 6, "0 1 2 3 4 5 6") will be called. Statement-1 is true for source 6. Therefore the string (0 1 2 3 4 5 6) will be printed and the function will be returned to printAllpath(5, 6, "0 1 2 3 4 5").

****

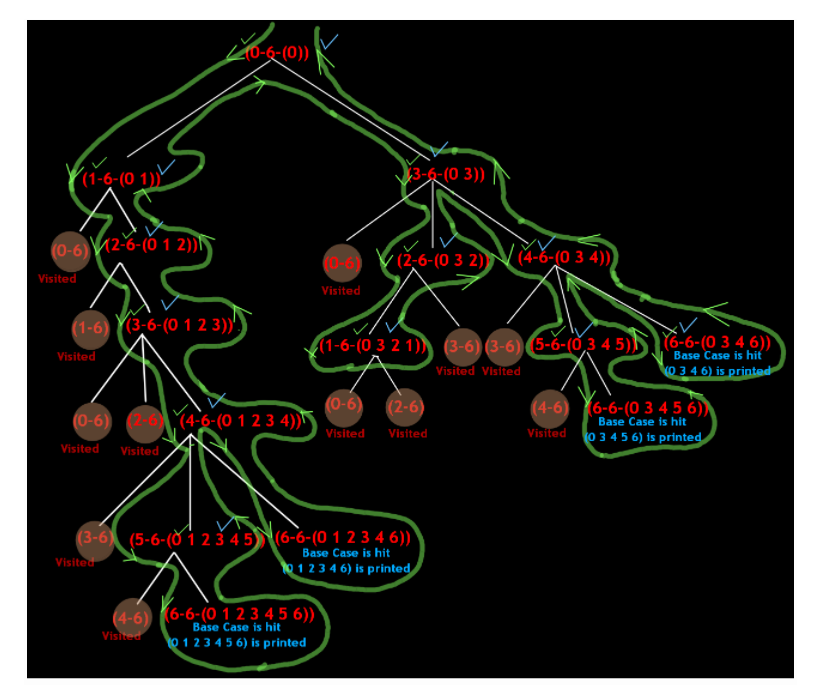
InprintAllPath(5, 6, "0 1 2 3 4 5"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 5 is set to false in the visited array. After that control returns to printAllPath(4, 6, "0 1 2 3 4"). In printAllPath(4, 6, "0 1 2 3 4"), since all the neighbors have not been explored, therefore a call is made to the next unexplored neighbor that is 6 in this case. > Then we check whether 6 has been visited earlier or not. 6 hasn't been visited, therefore printAllPath(6, 6, "0 1 2 3 4 6") will be called. Statement-1 is true for source 6. Therefore the string (0 1 2 3 4 6) will be printed and the function will be returned to printAllpath(4, 6, "0 1 2 3 4").> InprintAllPath(4, 6, "0 1 2 3 4"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 4 is set to false in the visited array.> After that control returns to printAllPath(3, 6, "0 1 2 3"). In printAllPath(3, 6, "0 1 2 3"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 3 is set to false in the visited array.> After that control returns to printAllPath(2, 6, "0 1 2"). In printAllPath(2, 6, "0 1 2"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 2 is set to false in the visited array. After that control returns to printAllPath(1, 6, "0 1"). inprintAllPath(1, 6, "0 1"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 1 is set to false in the visited array. After that control returns to printAllPath(0, 6, "0"). InprintAllPath(0, 6, "0"), since all the neighbors have not been explored, therefore a call is made to the next unexplored neighbor that is 3 in this case.

****

Then we check whether 3 has been visited or not. 3 hasn't been visited, therefore printAllPath(3 6, "0 3") will be called. Statement-1 is false for source 3. So control jumps to statement-2 and the value at index 3 is set true in the visited array. Then using a for loop, a recursive call to 3's neighbor is made. First neighbor is 0. Then we check whether 0 has been visited earlier or not. 0 has been visited, therefore we move to the next neighbor of 3 i.e. 2 in this case. Then we check whether 2 has been visited earlier or not. 2 has not been visited, therefore printAllPath(2, 6, "0 3 2") will be called. Statement-1 is false for source 2. So control jumps to statement-2 and the value at index 2 is set true in the visited array. Then using a for loop, a recursive call to 2's neighbor is made. First neighbor is 1. Then we check whether 1 has been visited earlier or not. 1 has not been visited, therefore printAllPath(1, 6, "0 3 2 1") will be called. Statement-1 is false for source 2. So control jumps to statement-2 and the value at index 2 is set true in the visited array. Then using a for loop, a recursive call to 2's neighbor is made. First neighbor is 1. Then using a for loop, a recursive call to 2's neighbor is made. First neighbor is 1. Statement-1 is false for source 1. So control jumps to statement-2 and the value at index 1 is set true in the visited array. Then using a for loop, a recursive call to 1's neighbor is made. First neighbor is 0. Then we check whether 0 has been visited earlier or not. 0 has been visited, therefore we move to the next neighbor of 1 i.e. 2 in this case. Then we check whether 2 has been visited earlier or not. 2 has also been visited. inprintAllPath(1, 6, "0 3 2 1"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 1 is set to false in the visited array. After that control returns to printAllPath(2, 6, "0 3 2"). inprintAllPath(2, 6, "0 3 2"), since all the neighbors have not been explored, therefore a call is made to the next unexplored neighbor that is 3 in this case. Then we check whether 3 has been visited or not. 3 has been visited. in printAllPath(2, 6, "0 3 2"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 2 is set to false in the visited array. After that control returns to printAllPath(3, 6, "0 3"). in printAllPath(3, 6, "0 3"), since all the neighbors have not been explored, therefore a call is made to the next unexplored neighbor that is 4 in this case therefore printAllPath(4, 6, "0 3 4") will be called. Statement-1 is false for source 4. So control jumps to statement-2 and the value at index 4 is set true in the visited array. Then using a for loop, a recursive call to 4's neighbor is made. First neighbor is 3. Then we check whether 3 has been visited earlier or not. 3 has been visited, therefore we move to the next neighbor of 4 i.e. 5 in this case. Then we check whether 5 has been visited earlier or not. 5 has not been visited, therefore printAllPath(5, 6, "0 3 4 5") will be called.

****

Statement-1 is false for source 5. So control jumps to statement-2 and the value at index 5 is set true in the visited array. Then using a for loop, a recursive call to 5's neighbor is made. First neighbor is 4. Then we check whether 4 has been visited earlier or not. 4 has been visited, therefore we move to the next neighbor of 5 i.e. 6 in this case. Then we check whether 6 has been visited earlier or not. 6 has not been visited, therefore printAllPath(6, 6, "0 3 4 5 6") will be called. Statement-1 is true for source 6.Therefore the string (0 3 4 5 6) will be printed and the function will be returned to printAllpath(5, 6, "0 3 4 5"). in printAllPath(5, 6, "0 3 4 5"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 5 is set to false in the visited array. After that control returns to printAllPath(4, 6, "0 3 4"). inprintAllPath(4, 6, "0 1 2 3 4"), since all the neighbors have not been explored, therefore a call is made to the next unexplored neighbor that is 6 in this case. Then we check whether 6 has been visited earlier or not. 6 hasn't been visited, therefore printAllPath(6, 6, "0 3 4 6") will be called. InprintAllPath(4, 6, "0 3 4 "), since all the neighbors have been explored, now control comes to statement-5 and the value at index 4 is set to false in the visited array. After that control returns to printAllPath(3, 6, "0 3"). printAllPath(3, 6, "0 3"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 3 is set to false in the visited array. After that control returns to printAllPath(0, 6, "0"). In printAllPath(0, 6, "0"), since all the neighbors have been explored, now control comes to statement-5 and the value at index 0 is set to false in the visited array.

****

**3. Code :**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList< Edge>[] graph = new ArrayList[vtces];

for (int i = 0; i < vtces; i++) {

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

for (int i = 0; i < edges; i++) {

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

int src = Integer.parseInt(br.readLine());

int dest = Integer.parseInt(br.readLine());

boolean[] visited = new boolean[vtces];

printAllPaths(graph, src, dest, visited, src + "");

}

public static void printAllPaths(ArrayList< Edge>[] graph, int src, int dest, boolean[] visited, String psf) {

if (src == dest) {

System.out.println(psf);

return;

}

visited[src] = true;

for (Edge e : graph[src]) {

if (!visited[e.nbr]) {

printAllPaths(graph, e.nbr, dest, visited, psf + e.nbr);

}

}

visited[src] = false;

}

}

**4. Analysis :**

Time Complexity:

O(V+E)

Space Complexity:

O(V)