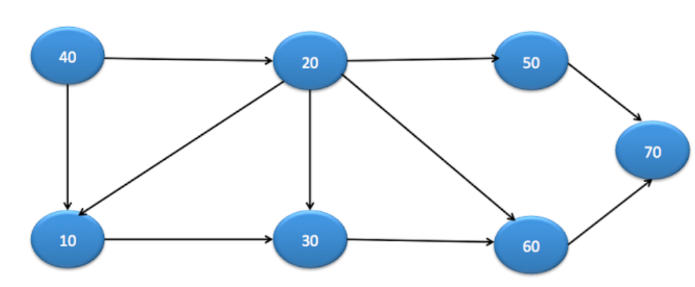
**Algorithm Implementation: Recursive Depth First Search Finite Graphs**

**Part 1 – Implementation:**



|  |
| --- |
| **public** **void** dfs(Node startNode){  System.out.println(startNode.data + “ “);  List<Node> visitedNodes = visitEligibleAdjacentNodes(startNode);  **for** (Node node : visitedNodes) {  dfs(node);  node.visited = **true**;  }  } |

**Part 2 – Execution Example:**

* Starting node is 40, eligible adjacent nodes are 10 and 20. We’ll start with 10 (since it was added to the list of nodes in the graph first). We will recurse with the 10 as our node in the parameter.
* The current node is now 10. The list of eligible adjacent nodes is only 30. We will recurse with the 30 as our node in the paramter.
* The current node is now 30. The list of eligible adjacent nodes is only 60. We will recurse with the 60 as our node in the paramter.
* The current node is now 60. The list of eligible adjacent nodes is only 70. We will recurse with the 70 as our node in the paramter.
* The current node is now 70. The list of eligible adjacent nodes is empty. We return from this recursive call.
* We back tracked to the 60. No more nodes to iterate. We return from this recursive call.
* We back tracked to the 30. No more nodes to iterate. We return from this recursive call.
* We back tracked to the 10. Now we have another iteration to do (the 20). We will recurse with 20 as our node in the paramter.
* The current node is now 20. The eligible adjacent nodes is 50. We will recurse with 50 as our node in the paramter.
* The current node is 50. Since there is no eligible adjacent nodes, we return from this recursive call.
* We backtracked to the 20. Since there is no more eligible adjacent nodes to iterate over, we return from this recursive call.
* We back tracked back to the 40 which is the start node. Since we iterated over all the adjacent nodes (the 10 and the 20) and recursed through all their adjacent nodes, this means we completely traversed the graph.

|  |
| --- |
| **Visual Representation:** (Note that CN = current node, EAN’s is eligible adjacent nodes, an indent to the right means we recursed, and indent left/undo indent means we backtracked.  ->CN:40 EAN's: 10,20,  ->CN:10 EAN's: 30,  ->CN:30 EAN's: 60,  ->CN:60 EAN's: 70,  ->CN:70 EAN's:  ->CN:20 EAN's: 50,  ->CN:50 EAN's: |

**Final Output:** 40,10,30,60,70,20,50

**Part 3 – Time and Space Complexity:**

|  |
| --- |
| **public** **void** dfs(Node startNode){  System.out.println(startNode.data + “ “);  List<Node> visitedNodes = visitEligibleAdjacentNodes(startNode);  **for** (Node node : visitedNodes) {  dfs(node);  node.visited = **true**;  }  } |

**Time Complexity:**

* Let’s ignore the function call “visit Eligible Adjacent Nodes”. Imagine we could do this in constant time. Then the complexity would be O(V). You can tell because there is one guranteed print for each time it enters the for loop, regardless of which recursive case this is. This is because once it enters the for loop, it will enter the “dfs” call which will immediately print that node.
* Another way to look at it is, is you add a constant amount of steps for each eligible adjacent nodes. The sum of all the eligible adjacent nodes is always going to be V. This is because each node can be eligible once (then it gets visited).
* A third way to look at is you are going to iterate over (the current node’s eligible adjacent nodes) + (the current nodes eligible adjacent nodes’s adjacent nodes). This is the same as iterating over V.
* So the time complexity is O(V + ∑ of all (visit eligible adjacent node calls)). This has been explained earlier to be O(V + E).

**Space Complexity:**

* The space complexity is the space it takes to store all the recursive call information. Remember that there is a print for each recursive calls, and there are V prints, so there are V recursive calls. This means it is O(V) to store the recursive calls.
* In addition, there is the space it takes store all the visited nodes. If you look at our graphs example, the node with the maximum amount of eligible adjacent nodes was the 40 (with two). So this list would store a maximum amount of 2 nodes. However, if the graph was orchaestrated differently, the max could have been V.
* Total is O(2V) = O(V)

**Summary:**

|  |  |
| --- | --- |
| **Time Complexity (Average and Worst Case)** | **Space Complexity** |
| ***O(V + E)*** | ***O(V)*** |