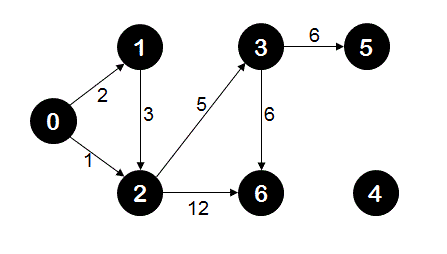
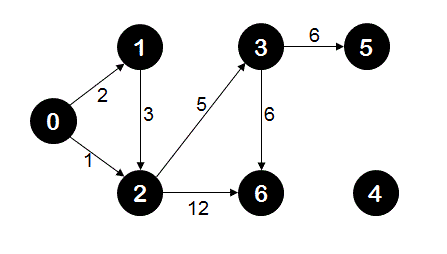
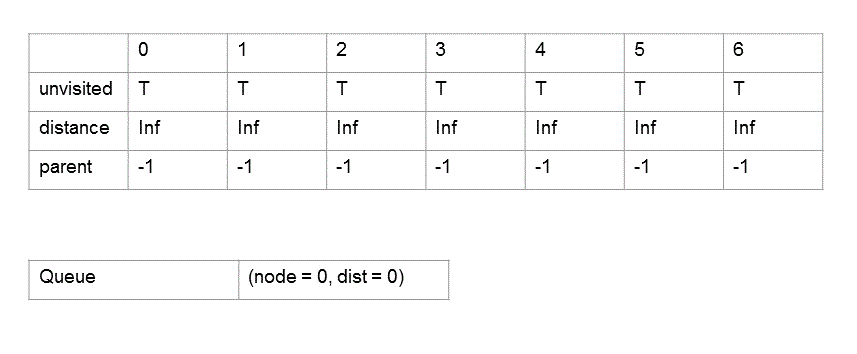
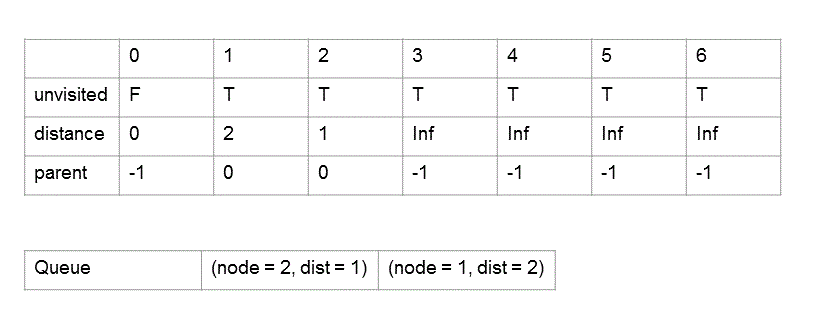
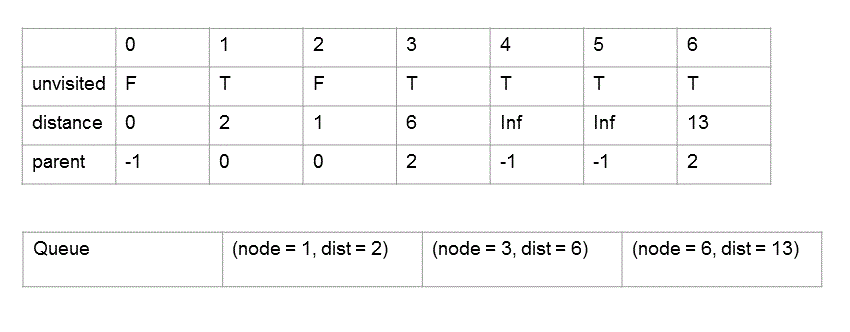
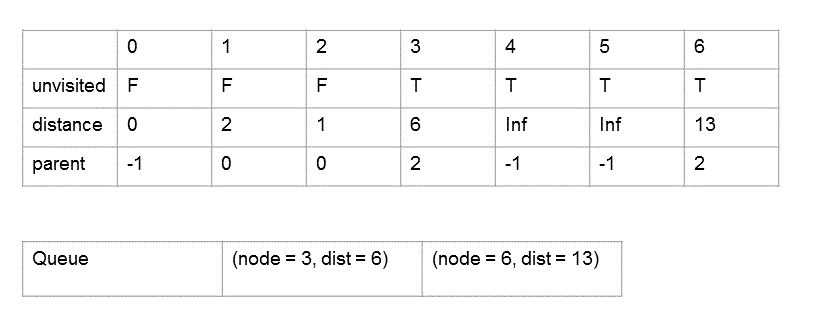
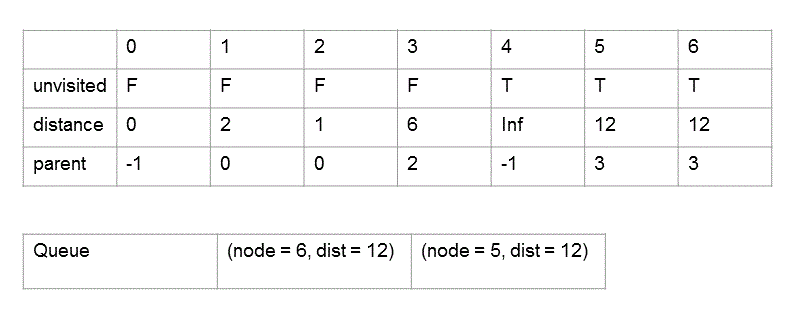
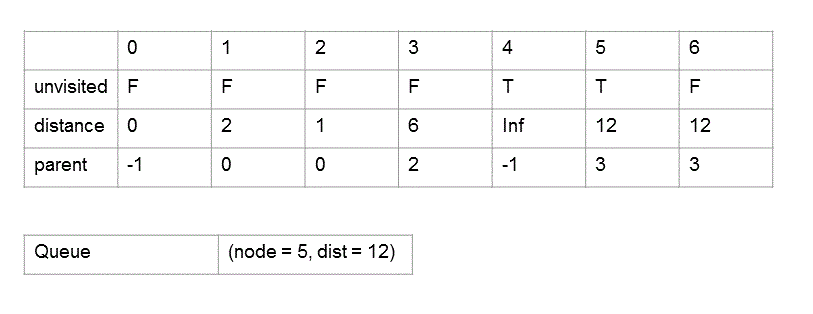
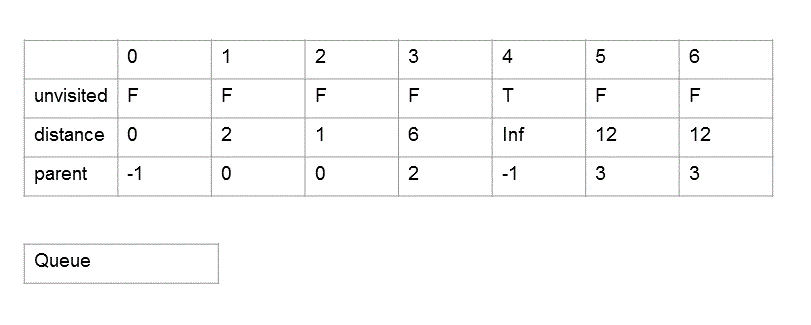
Dijkstra's Shortest Path algorithm

**Given a graph with no negative distance between any two nodes, find the shortest distance between initial node '0' and all nodes. For example, for the following graph  
    
Shortest distance of node '0' from node '0' is 0  
Shortest distance of node '1' from node '0' is 2  
Shortest distance of node '2' from node '0' is 1  
Shortest distance of node '3' from node '0' is 6  
Shortest distance of node '5' from node '0' is 12  
Shortest distance of node '6' from node '0' is 12  
and node '4' is unreachable from node '0'**

**Algorithm/Insights**

Dijkstra's algorithm essentially uses breadth first search with greedy approach to come up with the shortest distance between given two nodes.  
Let the node from which we would find the shortest distance of all other nodes be called initial node. We define the distance of node 'Y' as the distance from the initial node to node 'Y'. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step. The steps of the algorithm are as following  
  
1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.   
This step signifies that at the start of the algorithm, the starting node is at distance 0 from itself and other nodes are unreachable.  
  
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.  
  
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Also change the parent node of this neighbor as the current node. These parent nodes will help to backtrack the shortest path to a node from the source node. If the currently assigned distance value of the neighbor node is smaller than distance of neighbor from current node + current node's assigned distance then don't do anything.  
  
4. When we are done considering all the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.  
  
5. If there is no node remaining in unvisited set or the smallest distance of node in unvisited set is infinity then stop. Algorithm is completed.   
  
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.  
(Source: wikipedia)  
  
This algorithm is implemented using priority queue for keeping node with the shortest distance from the source at the front of the queue(remember greedy approach), parent array for keeping track of immediate parent of a node on the shortest path from source to that node, distance array to keep track of shortest distance of a node from source node, unvisted array. Let's see how it works for the graph shown here.  
  
  
1. To start with, all the nodes are unvisited, distance of all nodes from source node '0' is infinity and parent of all nodes is invalid/-1.  
  
2. Now we add source node '0' to the queue with distance as 0.  
  
  
3. We remove node 0 from queue, mark it as visited, make distance[0] as 0. Parent[0] still remains -1 since this is the starting vertex. We add neighbors of node 0 that is node 1 and node 2 to the queue, updated their parent to node 0, and update their distance entries as well.   
  
  
4. Now we remove node 2 from queue which would be at the front of the queue since it has the least distance of all nodes in the queue, mark this node as visited. We then add all the unvisited neighbors of node 2 (node 3 and node 6) to the queue if they are not already in the queue. Then for each unvisited neighbor of node 2 we check if the assigned distance of that neighbor is greater than assigned distance of node 2 + distance between node 2 and that neighbor. If this is the case then we update parent of that neighbor to node 2 and  update assigned distance of that neighbor to distance of node 2 + distance between node 2 and that neighbor.  
  
  
We repeat above step for all entries in the queue until the queue is not empty. As the algorithm progresses, the states of relevant data structures are shown below. At the end of the algorithm, each distance[i] specifies the shortest distance of node 'i' from node 0.  
  
5.   
  
6.   
  
7.   
  
8.   


Minimum number of trials to reach from source word to destination word

## Given a dictionary of words, find minimum number of trials to reach from source word to destination word. A valid trial on word 'w' is defined as either insert, delete or substitute operation of a single character in word 'w' which results in a word 'w1' which is also present in the given dictionary. For example, for dictionary {"BCCI","AICC","ICC","CCI","MCC","MCA", "ACC"}, minimum number of trials to reach from word "AICC" to "ICC" is 1. Only 1 opeartion of deleting character 'A' is required to reach from word "AICC" to word "ICC". Minimum number of trials to reach from "AICC" to "MCC" is 2(AICC->ICC->MCC) and minimum number of trials to reach from "AICC" to "MCA" is 3(AICC->ICC->MCC->MCA).  Now if you notice, there are no valid trials with source as "AICC" and destination as "BCCI" for above dictionary. Hence the output returned by program should be '-1' indicating destination word cannot be reached from source word.

## Video coming soon!

#### Subscribe for more updates

### Let's code now!

[Click to Code »](http://www.code.ideserve.co.in/)

# Algorithm/Insights

### The valid trial on a word is nothing but an edit operation consisting of either edit or delete or insert operation which results in another word present in given dictionary of words. Using this definition of valid trial, we can re-model this problem as a graph problem where nodes are created for representing all words in dictionary. If [edit distance](http://www.ideserve.co.in/learn/edit-distance-dynamic-programming)between two words 'word0' and 'word1' is 1, then an edge is added with cost as 1 between nodes 'n0' and 'n1' representing words 'word0' and 'word1' respectively.  Now to find out the minimum number of trials to reach from 'source' word to 'destination' word, all we need to do is to find out the minimum distance between node representing 'source' word and node representing 'destination' word. Because in this graph, all edges have equal cost of 1, doing a [breadth first search](http://www.ideserve.co.in/learn/breadth-first-search-in-graph)or depth first search for 'destination' node starting from 'source' node is sufficient to compute minimum distance between 'source' node and 'destination' node. This minimum distance obtained is nothing but the number of trials required to reach 'destination' word from 'source' word. Example: If we are asked to find out the minimum number of trials to reach from word "AICC" to word "MCA" for dictionary {"BCCI","AICC","ICC","CCI","MCC","MCA", "ACC"}, then we first create a graph with edges between those nodes which have edit distance of 1 between them. The graph created would look like following -    http://www.ideserve.co.in/learn/img/minTrials_0.gif Please note that this graph has two connected components where every node in a component is reachable from every other node. The edges in this graph are bi-directional and hence no direction arrows are shown. To find out the minimum number of trials to reach from "AICC" to "MCA", we simply start breadth first search from node "AICC" in above graph until we reach node "MCA". While doing breadth first search, starting node "AICC" is assumed to be at level-0, all its neighbors then would be at level-1, neighbors of immediate neighbor of "AICC" would be at level-2 and so on. While doing breadth first search, we keep track of the level of each node visited. During breadth first search, at level-0 only node "AICC" would be visited. At level-1, nodes "ACC" and "ICC" would be visited. Node "MCC" would be visited at level-2 and finally when node "MCA" is visited at level-3, its level that is level-3 is returned since this level would be equal to the minimum number of trials required to reach node "MCA" from node "AICC". The time complexity of this algorithm is O(m^2.n^2) where 'm' is total number of words in given dictionary and 'n' is the average length of each word. This is because while constructing graph, there are O(m^2) pairs of words for which edit distance is calculated in O(n^2) time. Note that time taken for breadth first search is O(|E|) = O(m^2) in the worst case which is less than time taken for graph building and hence this time is ignored while computing overall time complexity.