BFS

BFS

- Breadth First Search
- One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores the neighbor nodes first, before moving to the next level neighbors.
- Find minimum path/distance/...
  - Minimum Tree Depth
  - Word Edit Distance
  - etc...
- No Recursion, QUEUE

Queue

- Used for BFS
- FIFO
- VS Stack?
- Multitask Queue

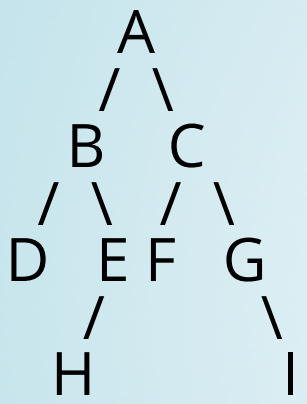
Queue in Java

- Queue<> queue = new LinkedList<>();
  - boolean offer(Element e); (add)
    - offer can return false while add can only cause exception when the element cannot be added due to capacity restriction.
  - Element poll(); (remove)
    - poll returns null when empty while remove will throw exception.
  - Element peek(); (element)
    - peek returns null when empty while element will throw exception.

BFS for Binary Trees

Binary Tree Level Order Traversal

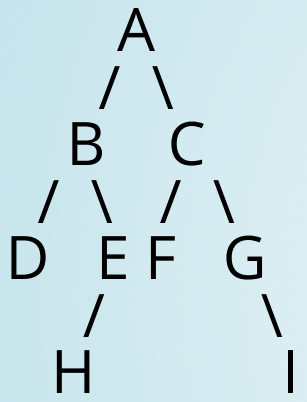
Given a binary tree, return its level order traversal.




```
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
while (!queue.isEmpty()) {
    TreeNode top = queue.poll();
    // visit top.
    if (top.left != null) {
        queue.offer(top.left);
    }
    if (top.right != null) {
        queue.offer(top.right);
    }
}
```

Binary Tree Level Order Traversal

Given a binary tree, return its level order traversal.

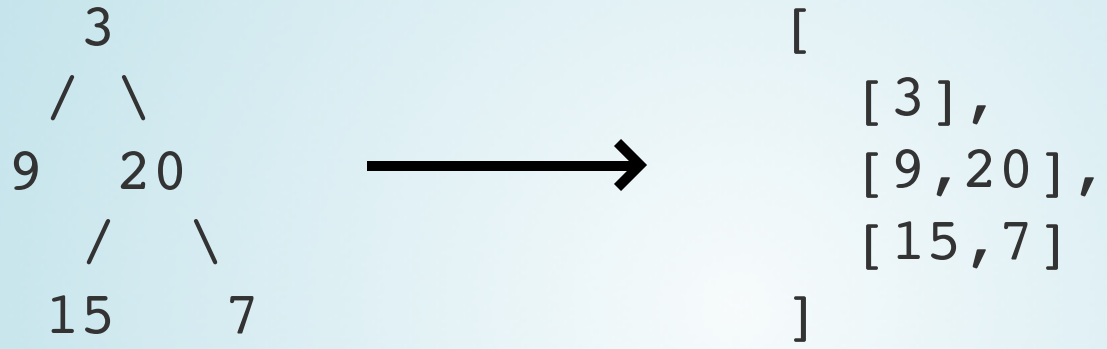


The output is A B C D E F G H I

What if we want to separate each level?

Binary Tree Level Order Traversal

Given a binary tree, return its level order traversal.



Binary Tree Level Order Traversal

Given a binary tree, return its level order traversal.

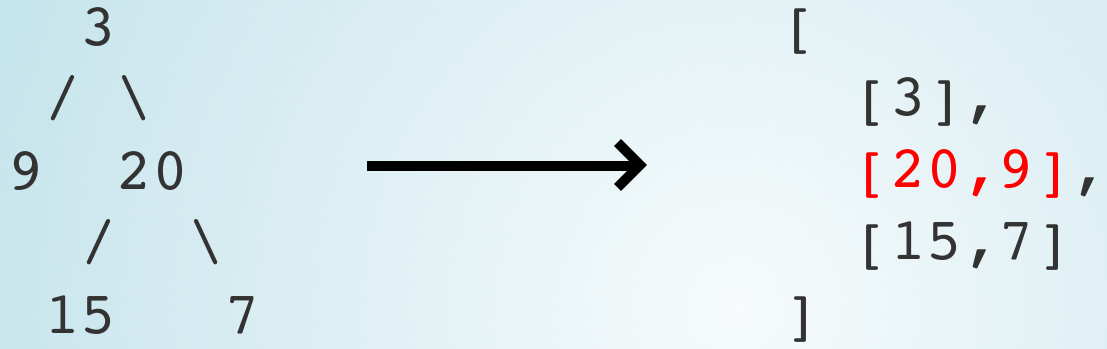
```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> results = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    if (root != null) {
        queue.offer(root);
    }
    while (!queue.isEmpty()) {
        List<Integer> oneResult = new ArrayList<>();
        Queue<TreeNode> queue2 = new LinkedList<>();
        while (!queue.isEmpty()) {
            TreeNode top = queue.poll();
            if (top.left != null) {
                queue2.offer(top.left);
            }
            if (top.right != null) {
                queue2.offer(top.right);
            }
            oneResult.add(top.val);
        }
        results.add(oneResult);
        queue = queue2;
    }
    return results;
}
```

Binary Tree Level Order Traversal

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> results = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    queue.offer(null);
    List<Integer> oneResult = new ArrayList<>();
    while (!queue.isEmpty()) {
        TreeNode top = queue.poll();
        if (top == null) {
            results.add(oneResult);
            if (!queue.isEmpty()) {
                queue.offer(null);
            }
            oneResult = new ArrayList<>();
        } else {
            if (top.left != null) {
                queue.offer(top.left);
            }
            if (top.right != null) {
                queue.offer(top.right);
            }
            oneResult.add(top.val);
        }
    }
    return results;
}
```

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return its zigzag level order traversal.



Binary Tree Zigzag Level Order Traversal

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> results = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    if (root != null) {
        queue.offer(root);
    }
    boolean isOdd = true;
    while (!queue.isEmpty()) {
        List<Integer> oneResult = new ArrayList<>();
        Queue<TreeNode> queue2 = new LinkedList<>();
        while (!queue.isEmpty()) {
            TreeNode top = queue.poll();
            if (top.left != null) {
                queue2.offer(top.left);
            }
            if (top.right != null) {
                queue2.offer(top.right);
            }
            oneResult.add(top.val);
        }
        if (!isOdd) {
            Collections.reverse(oneResult);
        }
        results.add(oneResult);
        isOdd = !isOdd;
        queue = queue2;
    }
    return results;
}
```

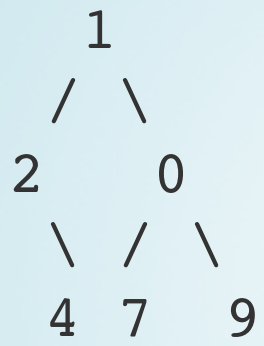
Binary Tree Zigzag Level Order Traversal

- Reversing an ArrayList for every loop is time-consuming.
- Visit order is opposite from level to level
  - LIFO
  - Stack

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> results = new ArrayList<>();
    if (root == null)
        return results;
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    boolean isOdd = true;
    while (!stack.isEmpty()) {
        List<Integer> oneResult = new ArrayList<>();
        Stack<TreeNode> stack2 = new Stack<>();
        while (!stack.isEmpty()) {
            TreeNode top = stack.pop();
            if (!isOdd) {
                if (top.right != null)
                    stack2.push(top.right);
                if (top.left != null)
                    stack2.push(top.left);
            } else {
                if (top.left != null)
                    stack2.push(top.left);
                if (top.right != null)
                    stack2.push(top.right);
            }
            oneResult.add(top.val);
        }
        results.add(oneResult);
        stack = stack2;
        isOdd = !isOdd;
    }
    return results;
}
```

Find Bottom Left Tree Value

Given a binary tree, find the leftmost value in the last row of the tree.



Result: 4

BFS

- Same as level traversal, but need to know which one is the first one of this level

Find Bottom Left Tree Value

```
public int findBottomLeftValue(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    int leftValue = 0;
    boolean isFirst = false;
    while (!queue.isEmpty()) {
        isFirst = true;
        Queue<TreeNode> queue2 = new LinkedList<TreeNode>();
        while (!queue.isEmpty()) {
            TreeNode top = queue.poll();
            if (isFirst) {
                leftValue = top.val;
                isFirst = false;
            }
            if (top.left == null && top.right == null) {
                continue;
            }
            if (top.left != null) {
                queue2.offer(top.left);
            }
            if (top.right != null) {
                queue2.offer(top.right);
            }
        }
        queue = queue2;
    }
    return leftValue;
}
```

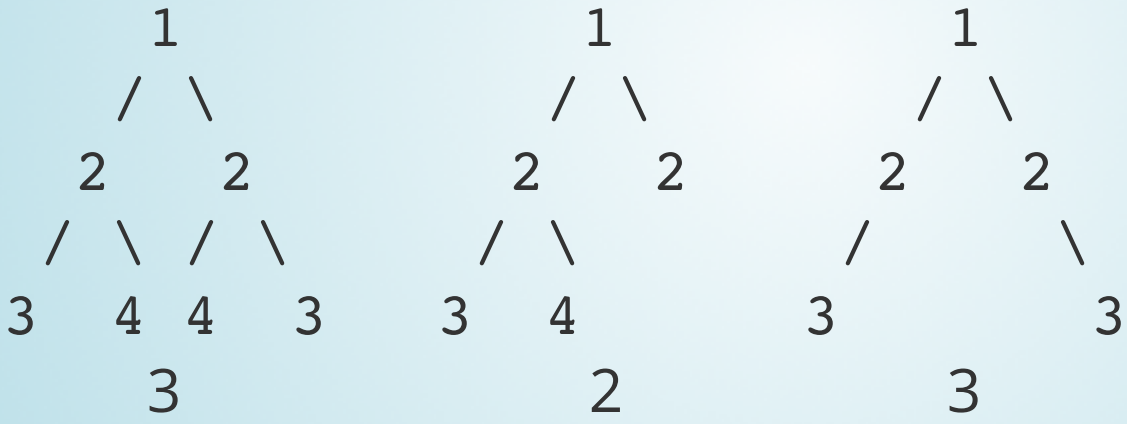
How to level order traverse

- If level is important, most cases
  - Two queues.
  - Dummy node, different from all other nodes to be a flag.
  - Maintain size of current level.
- If level is unimportant
  - One queue.

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.



Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

DFS

- Minimum depth of Root
  - if root is leaf, 1.
  - if root.left is null, return right.
  - if root.right is null, return left.
  - return the minimum of (left, right).

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

BFS

- Level Traverse the tree, return the depth of the first leaf node.

Minimum Depth of Binary Tree

```
public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    queue.offer(null);
    int minDepth = 1;
    while (!queue.isEmpty()) {
        TreeNode top = queue.poll();
        if (top == null) {
            minDepth++;
            queue.offer(null);
            continue;
        }
        if (top.left == null && top.right == null) {
            return minDepth;
        }
        if (top.left != null) {
            queue.offer(top.left);
        }
        if (top.right != null) {
            queue.offer(top.right);
        }
    }
    return minDepth;
}
```

Minimum Depth of Binary Tree

```
public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    int minDepth = 1;
    while (!queue.isEmpty()) {
        Queue<TreeNode> queue2 = new LinkedList<TreeNode>();
        while (!queue.isEmpty()) {
            TreeNode top = queue.poll();
            if (top.left == null && top.right == null) {
                return minDepth;
            }
            if (top.left != null) {
                queue2.offer(top.left);
            }
            if (top.right != null) {
                queue2.offer(top.right);
            }
        }
        minDepth++;
        queue = queue2;
    }
    return minDepth;
}
```

BFS for Binary Trees

- Level is important, most cases
  - Two queues.
  - Dummy node, different from all other nodes to be a flag.
  - Maintain size of current level.
- Level is unimportant
  - One queue.
- Don't add null (except dummy) into queues.
- Left/Right null pointer check.

General BFS

The Maze II

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

The Maze II

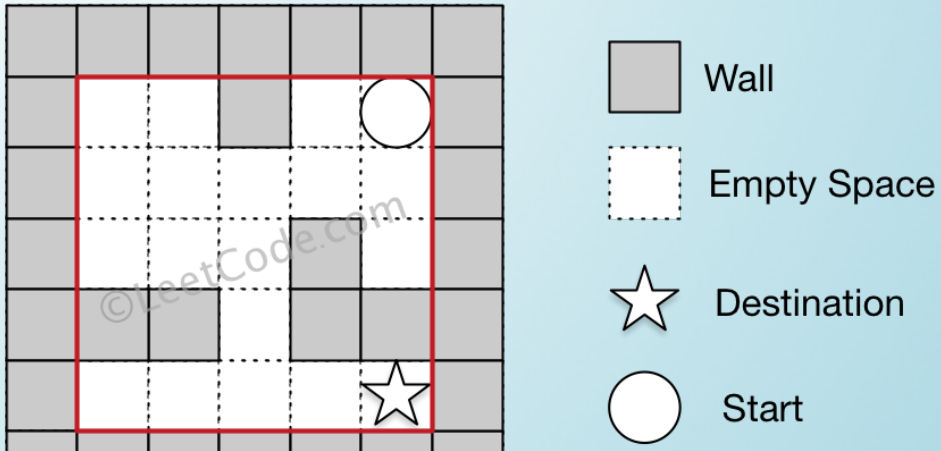
Example 1
Input 1: a maze represented by a 2D array

0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: 12

Explanation: One shortest way is : left -> down -> left -> down -> right -> down -> right. The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.



The Maze II

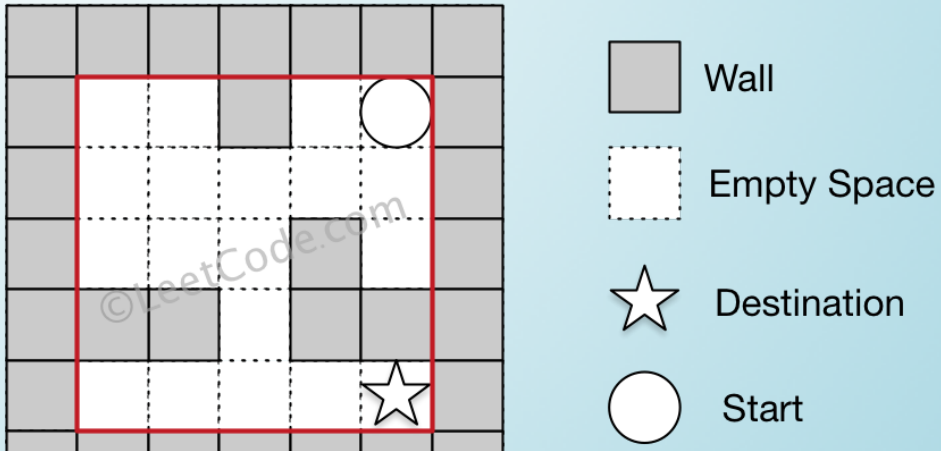
We need to use BFS to solve the problem since it wants to get the shortest way. And BFS could simulate the behavior well.

The most important thing is to let the queue mock the action that the ball will continue rolling till it hits the edge or a block.

As long as we can mock this operation, it will be much easier for us to solve the problem.

How do we simulate this operation?

For each step we pop from queue, we need to use a while loop to keep rolling the ball till the end



```
public int findShortestWay(int[][] maze, int[] start, int[] dest) {
    int row = maze.length;
    int col = maze[0].length;
    int[][] distance = new int[row][col];
    for (int i = 0; i < row; i ++) {
        for (int j = 0; j < col; j ++) {
            distance[i][j] = Integer.MAX_VALUE;
        }
    }
    int[] dx = new int[] {1, 0, 0, -1};
    int[] dy = new int[] {0, -1, 1, 0};
    Queue<Pair> queue = new LinkedList<Pair>();
    queue.offer(new Pair(start[0],start[1]));
    distance[start[0]][start[1]] = 0;
    while (!queue.isEmpty()) {
        Pair current = queue.poll();
        for (int i = 0; i < 4; i ++) {
            int x = current.x;
            int y = current.y;
            int dist = distance[x][y];
            while (x >= 0 && x < row && y >= 0 & y < col && maze[x][y] == 0) {
                x += dx[i];
                y += dy[i];
                dist ++;
            }
            x -= dx[i];
            y -= dy[i];
            --dist;
            if (distance[x][y] > dist) {
                distance[x][y] = dist;
                if (x != dest[0] || y != dest[1]) {
                    queue.offer(new Pair(x, y));
                }
            }
        }
    }
    int res = distance[dest[0]][dest[1]];
    return res == Integer.MAX_VALUE ? -1 : res;
}
```

```
class Pair {
    int x;
    int y;
    public Pair(int a, int b) {
        x = a;
        y = b;
    }
}
```

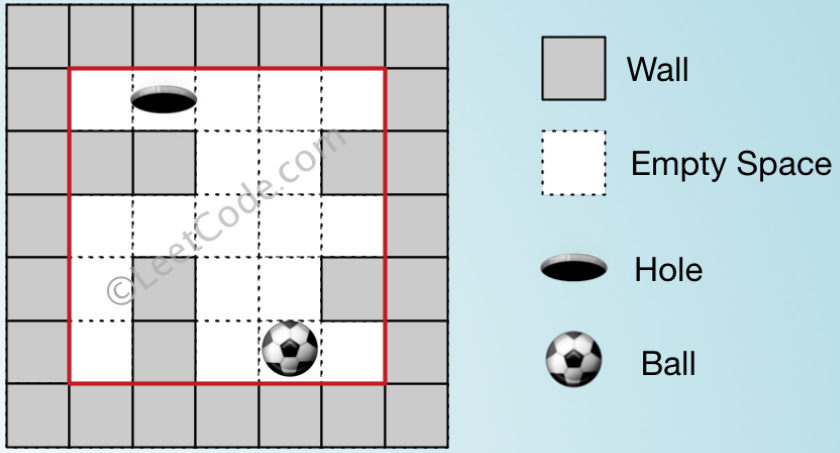
The Maze III

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up** (u), **down** (d), **left** (l) or **right** (r), but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction. There is also a **hole** in this maze. The ball will drop into the hole if it rolls on to the hole.

Given the **ball position**, the **hole position** and the **maze**, your job is to find out how the ball could drop into the hole by moving **shortest distance** in the maze. The distance is defined by the number of **empty spaces** the ball go through from the start position (exclude) to the hole (include). Output the moving **directions** by using 'u', 'd', 'l' and 'r'. Since there may have several different shortest ways, you should output the **lexicographically smallest** way. If the ball cannot reach the hole, output "impossible".

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The ball and hole coordinates are represented by row and column indexes.

The Maze III



Example:

Input 1: a maze represented by a 2D array

0 0 0 0 0
1 1 0 0 1
0 0 0 0 0
0 1 0 0 1
0 1 0 0 0

Input 2: ball coordinate (rowBall, colBall) = (4, 3)
Input 3: hole coordinate (rowHole, colHole) = (0, 1)

Output: "lul"
Explanation: There are two shortest ways for the ball to drop into the hole.
The first way is left -> up -> left, represented by "lul".
The second way is up -> left, represented by 'ul'.
Both ways have shortest distance 6, but the first way is lexicographically smaller because 'l' < 'u'. So the output is "lul".

The Maze III

We still need to use BFS to solve the problem.

We need to use a Path to keep the result since even within the same shortest way, we want to output the smallest one in lexicographical order.

The other part is very similar to The Maze II. But one thing to keep in mind:

If you hit the destination, since it is a trap, you can stop immediately instead of keeping rolling foward.

The Maze III

```
public String findShortestWay(int[][] maze, int[] ball, int[] hole) {
    int row = maze.length;
    int col = maze[0].length;
    int[] dx = new int[] {1, 0, 0, -1};
    int[] dy = new int[] {0, -1, 1, 0};
    String[] dir = new String[] {"d", "l", "r", "u"};
    int[][] distance = new int[row][col];
    for (int i = 0; i < row; i ++) {
        for (int j = 0; j < col; j ++) {
            distance[i][j] = Integer.MAX_VALUE;
        }
    }
    distance[ball[0]][ball[1]] = 0;
    Queue<Pair> queue = new LinkedList<Pair>();
    queue.offer(new Pair(ball[0], ball[1]));
    String[][] path = new String[row][col];
    path[ball[0]][ball[1]] = "";
```

The Maze III

```
    while(!queue.isEmpty()) {
        Pair cur = queue.poll();
        System.out.println(cur.x + " " + cur.y);
        for(int i = 0; i < 4; i++) {
            int x = cur.x, y = cur.y;
            int dist = distance[x][y];
            String currentPath = path[x][y];
            while(x >= 0 && x < row && y >= 0 && y < col && maze[x][y] == 0
                    && (x != hole[0] || y != hole[1])) {
                x += dx[i];
                y += dy[i];
                dist ++;
            }
            if (x != hole[0] || y != hole[1]) {
                x -= dx[i];
                y -= dy[i];
                dist --;
            }
            String newPath = currentPath.concat(dir[i]);
            if (distance[x][y] > dist || (distance[x][y] == dist
                && path[x][y].compareTo(newPath) > 0)) {
                distance[x][y] = dist;
                path[x][y] = newPath;
                if (x != hole[0] || y != hole[1]) queue.offer(new Pair(x, y));
            }
        }
    }
    String result = path[hole[0]][hole[1]];
    return distance[hole[0]][hole[1]] == Integer.MAX_VALUE ? "impossible" : result;
}
```

Word Ladder

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

- Only one letter can be changed at a time
- Each intermediate word must exist in the word list

For example,
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog", return its length 5.

Note:

- Return 0 if there is no such transformation sequence.
- All words have the same length.
- All words contain only lowercase alphabetic characters.

Word Ladder

DFS

- begin: hot, end: hip, list: [hog, hig, dig, dip, hip, hit]
- hot -> hog -> hig -> dig -> dip -> hip
- Much deeper than needed, time consuming.

BFS

- Keep a set of all words that can be transformed to.
- Put all words with 1 edit distance into the set every time.
- When the end word is met, then the minimum distance is found.

Word Ladder

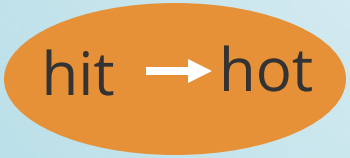
- BFS
  - Keep a set of all words that can be transformed to.
  - Put all words with 1 edit distance into the set every time.
  - When the end word is met, then the minimum distance is found.

beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

Word Ladder

- BFS
  - Keep a set of all words that can be transformed to.
  - Put all words with 1 edit distance into the set every time.
  - When the end word is added, then the minimum distance is found.

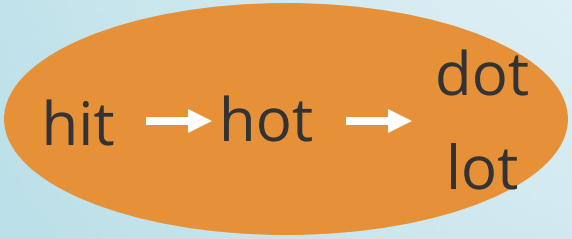
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]



Word Ladder

- BFS
  - Keep a set of all words that can be transformed to.
  - Put all words with 1 edit distance into the set every time.
  - When the end word is added, then the minimum distance is found.

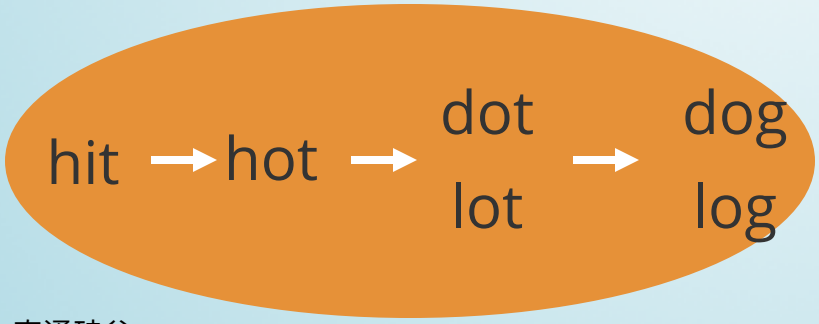
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]



Word Ladder

- BFS
  - Keep a set of all words that can be transformed to.
  - Put all words with 1 edit distance into the set every time.
  - When the end word is added, then the minimum distance is found.

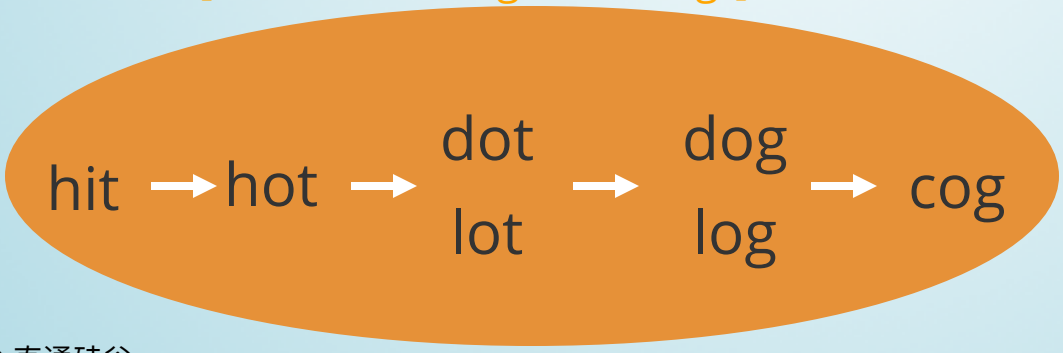
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]



Word Ladder

- BFS
  - Keep a set of all words that can be transformed to.
  - Put all words with 1 edit distance into the set every time.
  - When the end word is added, then the minimum distance is found.

beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]



Word Ladder

```
transformed_words <- beginWord
while (transformed_words is not empty) {
    for (word_b in transformed_words) {
        if (endWord is within 1 distance) {
            return distance
        }
        if (some word in dict is within 1 distance
             && it is not in transformed_words) {
            transformed_words.add(word)
        }
    }
    update distance
}
return 0
```

How to check?

Word Ladder

```
transformed_words <- beginWord
while (transformed_words is not empty) {
    for (word_b in transformed_words) {
        if (endWord is within 1 distance) {
            return distance
        }
        if (some word in dict is within 1 distance
            && it is not in transformed_words) {
            transformed_words.add(word)
        }
    }
    update distance
}
return 0
```

How to check?

Word Ladder

```
transformed_words <- beginWord
while (transformed_words is not empty) {
    for (word_b in transformed_words) {
        if (endWord is within 1 distance) {
            return distance
        }
        if (some word in dict is within 1 distance
             && it is not in transformed_words) {
            transformed_words.add(word)
        }
    }
    update distance
}
return 0
```

How to check?

Word Ladder

```
public int ladderLength(String beginWord, String endWord, List<String> wordList)
    Set<String> wordSet = new HashSet<>();
    for(String aWord: wordList) {
        wordSet.add(aWord);
    }
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    Set<String> visited = new HashSet<>();
    visited.add(beginWord);
    int distance = 1;
    while (!queue.isEmpty()) {
        Queue<String> queue2 = new LinkedList<>();
        distance++;
        while (!queue.isEmpty()) {
            String top = queue.poll();
            List<String> wordsWithinDistance =
                getWordsWithinDistance(wordSet, top);
            for (String word : wordsWithinDistance) {
                if (word.equals(endWord)) {
                    return distance;
                }
                if (!visited.contains(word)) {
                    queue2.add(word);
                    visited.add(word);
                }
            }
        }
        queue = queue2;
    }
    return 0;
}
```

Word Ladder

```
public ArrayList<String> getWordsWithinDistance(Set<String> wordSet, String word
    ArrayList<String> results = new ArrayList<>();
    char[] wordCharArr = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char oriChar = wordCharArr[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c == oriChar) {
                continue;
            }
            wordCharArr[i] = c;
            String newStr = new String(wordCharArr);
            if (wordSet.contains(newStr)) {
                results.add(newStr);
            }
        }
        wordCharArr[i] = oriChar;
    }
    return results;
}
```

Word Ladder

```
public int ladderLength(String beginWord, String endWord,
                        List<String> wordList) {
    Set<String> wordSet = new HashSet<>();
    for(String aWord: wordList) {
        wordSet.add(aWord);
    }
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    int distance = 1;
    while (!queue.isEmpty() && !wordSet.isEmpty()) {
        Queue<String> queue2 = new LinkedList<>();
        distance++;
        while (!queue.isEmpty()) {
            String top = queue.poll();
            ArrayList<String> wordsWithinDistance =
                getWordsWithinDistance(wordSet, top);
            if (wordsWithinDistance.contains(endWord)) {
                return distance;
            }
            queue2.addAll(wordsWithinDistance);
        }
        queue = queue2;
    }
    return 0;
}
```

Word Ladder

```
public ArrayList<String> getWordsWithinDistance(Set<String> wordSet,
                                                String word) {
    ArrayList<String> results = new ArrayList<>();
    char[] wordCharArr = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char oriChar = wordCharArr[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c == oriChar) {
                continue;
            }
            wordCharArr[i] = c;
            String newStr = new String(wordCharArr);
            if (wordSet.contains(newStr)) {
                results.add(newStr);
                wordSet.remove(newStr);
            }
        }
        wordCharArr[i] = oriChar;
    }
    return results;
}
```

Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]

Return

```
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]

Word Ladder II

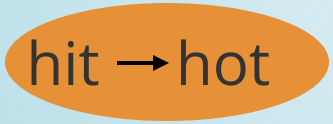
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



Word Ladder II

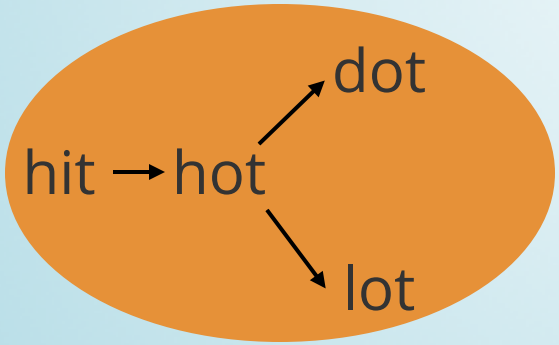
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



Word Ladder II

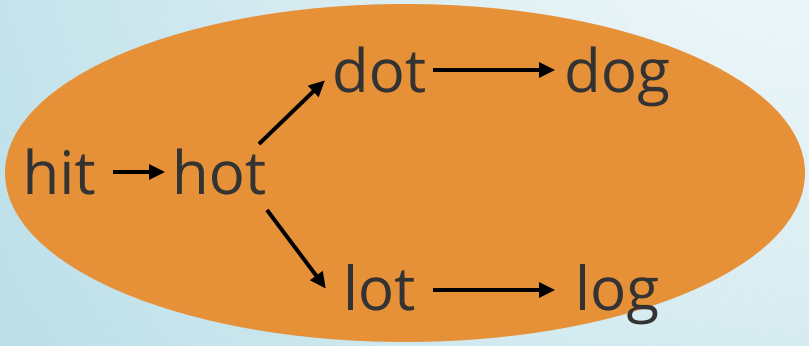
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



Word Ladder II

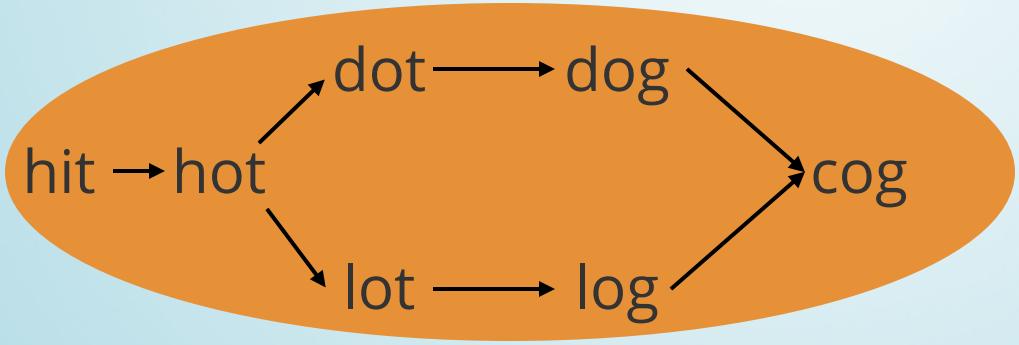
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



Word Ladder II

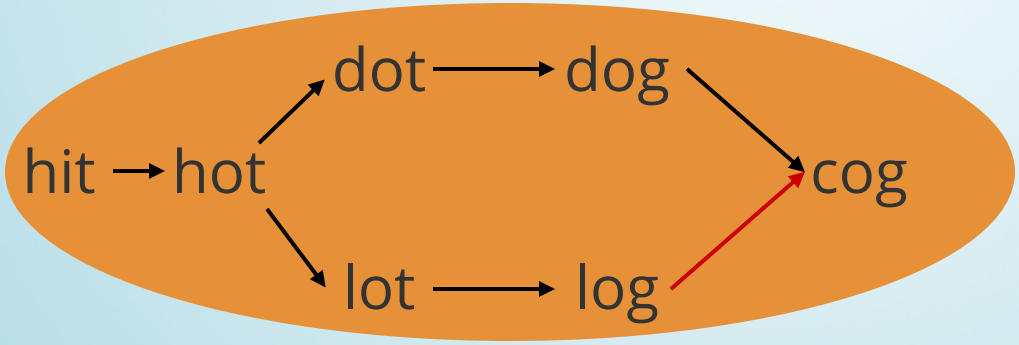
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



- Visited needs to be updated after one level.
- wordList need to be updated after one level.

Word Ladder II

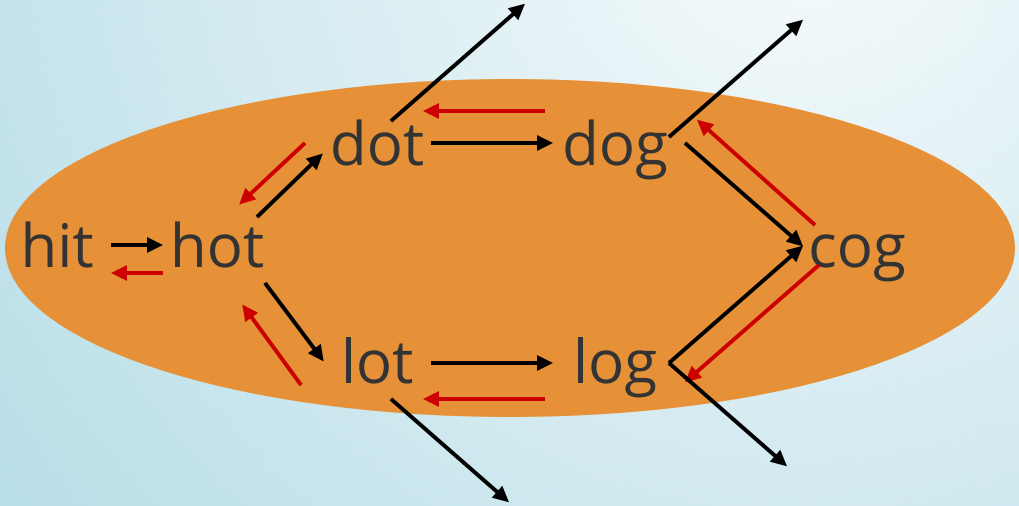
Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

Example:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log"]



- Path needs to be recorded backwards.
- HashMap<String, ArrayList<String>>

Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

- BFS to get shortest transformation
- Record the path information during transformation so that the sequence can get regenerated later.
  - with visited set, all nodes on the path can guarantee the shortest distance.
- DFS to get all the transformation sequences
  - backward from endWord instead of forward from startWord

Word Ladder II

```
public List<List<String>> findLadders(String beginWord, String endWord,
                                      List<String> wordList) {
    Set<String> wordSet = new HashSet<>();
    for(String aWord: wordList) {
        wordSet.add(aWord);
    }
    List<List<String>> results = new ArrayList<>();
    HashMap<String, ArrayList<String>> preList = new HashMap<>();

    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    Set<String> visited = new HashSet<>();
    visited.add(beginWord);
```

Word Ladder II

```
    while (!queue.isEmpty() && !wordList.isEmpty()) {
        Set<String> queue2 = new HashSet<>();
        boolean isFinished = false;
        while (!queue.isEmpty()) {
            String top = queue.poll();
            Set<String> wordsWithinDistance =
                getWordsWithinDistance(wordList, top);

            for (String word : wordsWithinDistance) {
                if (word.equals(endWord)) {
                    isFinished = true;
                }
                if (!visited.contains(word)) {
                    updatePreList(preList, word, top);
                    queue2.add(word);
                }
            }
        }
        visited.addAll(queue2);
        if (isFinished) {
            getPaths(results, preList, new ArrayList<String>(), endWord);
            break;
        }
        queue.addAll(queue2);
    }
    return results;
}
```

Word Ladder II

```
public void updatePreList(HashMap<String, ArrayList<String>> preList,
                          String cur, String pre) {
    if (!preList.containsKey(cur)) {
        preList.put(cur, new ArrayList<String>());
    }
    preList.get(cur).add(pre);
}

public Set<String> getWordsWithinDistance(Set<String> wordSet, String word) {
    Set<String> results = new HashSet<>();
    char[] wordCharArr = word.toCharArray();
    for (int i = 0; i < word.length(); i++) {
        char oriChar = wordCharArr[i];
        for (char c = 'a'; c <= 'z'; c++) {
            if (c == oriChar) {
                continue;
            }
            wordCharArr[i] = c;
            String newStr = new String(wordCharArr);
            if (wordSet.contains(newStr)) {
                results.add(newStr);
            }
        }
        wordCharArr[i] = oriChar;
    }
    return results;
}
```

Word Ladder II

- How to get paths backward from endWord?
  - DFS (recursion)
    - Base Case: endWord.pre == null
    - Recursion rule: path from endWord = path from endWord.pre + endWord

Word Ladder II

```
public void getPaths(List<List<String>> paths,
                     HashMap<String, ArrayList<String>> preList,
                     List<String> curPath,
                     String end) {
    if (!preList.containsKey(end)) {
        curPath.add(end);
        Collections.reverse(curPath);
        paths.add(curPath);
        return;
    }
    for (String pre : preList.get(end)) {
        List<String> newPath = new ArrayList<String>(curPath);
        newPath.add(end);
        getPaths(paths, preList, newPath, pre);
    }
}
```

```java
public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
    Set<String> wordSet = new HashSet<>();
    for(String aWord: wordList) {
        wordSet.add(aWord);
    }
    wordSet.remove(beginWord);
    List<List<String>> results = new ArrayList<>();
    HashMap<String, ArrayList<String>> preList = new HashMap<>();
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);

    while (!queue.isEmpty() && !wordList.isEmpty()) {
        Set<String> queue2 = new HashSet<>();
        boolean isFinished = false;
        while (!queue.isEmpty()) {
            String top = queue.poll();
            Set<String> wordsWithinDistance = getWordsWithinDistance(wordSet, top);
            for (String word : wordsWithinDistance) {
                if (word.equals(endWord)) {
                    isFinished = true;
                }
                updatePreList(preList, word, top);
                queue2.add(word);
            }
        }
        wordSet.removeAll(queue2);
        if (isFinished) {
            getPaths(results, preList, new ArrayList<String>(), endWord);
            break;
        }
        queue.addAll(queue2);
    }
    return results;
}
```

```
public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
    List<List<String>> results = new ArrayList<>();
    HashMap<String, ArrayList<String>> preList = new HashMap<>();
    Set<String> wordSet = new HashSet<>();
    for(String aWord: wordList) {
        wordSet.add(aWord);
    }
    wordSet.remove(beginWord);
    Queue<String> queue = new LinkedList<>();
    queue.offer(beginWord);
    while (!queue.isEmpty() && !wordList.isEmpty()) {
        Set<String> queue2 = new HashSet<>();
        boolean isFinished = false;
        while (!queue.isEmpty()) {
            String top = queue.poll();
            Set<String> wordsWithinDistance = getWordsWithinDistance(wordSet, top);
            updatePreList(preList, wordsWithinDistance, top);
            queue2.addAll(wordsWithinDistance);
            isFinished = isFinished | wordsWithinDistance.contains(endWord);
        }
        wordSet.removeAll(queue2);
        if (isFinished) {
            getPaths(results, preList, new ArrayList<String>(), endWord);
            break;
        }
        queue.addAll(queue2);
    }
    return results;
}
public void updatePreList(HashMap<String, ArrayList<String>> preList,
                          Set<String> curStrings, String pre) {
    for (String cur : curStrings) {
        if (!preList.containsKey(cur)) {
            preList.put(cur, new ArrayList<String>());
        }
        preList.get(cur).add(pre);
    }
}
```

Word Ladder II

Given two words (*beginWord* and *endWord*), and a dictionary's word list, find all shortest transformation sequence(s) from *beginWord* to *endWord*

- BFS to get shortest transformation
- Record the paths from beginning while BFSing
- Add only the valid paths into results.

BFS for Word Distance

- Distance means level important
  - Two queues
    - Order is not important, so queue is not essential actually.
  - Dummy node
- Pruning for duplicate word
  - HashSet for visited words
  - Remove from dict.
- Spell check
  - Word, Google Docs, etc.

BFS for Distance

- Shortest path
  - Dijkstra's Algorithm
- Maze solving
  - The minimum steps needed

DFS v.s. BFS

- Both complete search
- DFS is recursion
  - stack
  - better at checking connection, etc.
- BFS is iteration
  - queue
  - better at minimum distance, etc.
- Many problems are both solvable by DFS/BFS
  - Surrounded Region
  - Number of Islands
  - etc.

Surrounded Region

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.
A region is captured by flipping all 'O's into 'X's in that surrounded region.

Surrounded Region

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

```
public void solve(char[][] board) {
    if (board.length == 0 || board[0].length == 0)
        return;

    for (int j = 0; j < board[0].length; j++) {
        search(board, 0, j);
        search(board, board.length-1, j);
    }
    for (int i = 0; i < board.length; i++) {
        search(board, i, 0);
        search(board, i, board[0].length-1);
    }

    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            board[i][j] = board[i][j] == 'F' ? 'O' : 'X';
        }
    }
}
```

Surrounded Region

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.
A region is captured by flipping all 'O's into 'X's in that surrounded region.

```
// BFS
public void search(char[][] board, int x, int y) {
    if (board[x][y] == 'X') {
        return;
    }
    Queue<Integer> queue = new LinkedList<Integer>();
    int xLen = board.length, yLen = board[0].length;
    int[] dx = {-1, 0, 1, 0};
    int[] dy = {0, 1, 0, -1};

    queue.offer(x * yLen + y);
    board[x][y] = 'F';

    while (!queue.isEmpty()) {
        int temp = queue.poll();
        for (int i = 0; i < 4; i++) {
            int nx = temp / yLen + dx[i], ny = temp % yLen + dy[i];
            if (nx >= 0 && nx < xLen && ny >= 0 && ny < yLen
                && board[nx][ny] == 'O') {
                board[nx][ny] = 'F';
                queue.offer(nx * yLen + ny);
            }
        }
    }
}
```

Surrounded Region

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'.

A region is captured by flipping all 'O's into 'X's in that surrounded region.

```
// DFS
public void search(char[][] board, int i, int j) {
    if (board[i][j] != 'O')
        return;
    board[i][j] = 'F';
    if (i > 1)
        search(board, i-1, j);
    if (i < board.length - 2)
        search(board, i+1, j);
    if (j > 1)
        search(board, i, j-1);
    if (j < board[i].length - 2)
        search(board, i, j+1);
}
```

Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses (and).

We talk about this problem using DFS

How to use BFS?

Starting from the longest, generate all possible subString and push them into the queue

Remove Invalid Parentheses

```
public List<String> removeInvalidParentheses(String s) {
    List<String> result = new ArrayList<>();
    Queue<String> queue = new LinkedList<>();
    Set<String> checked = new HashSet<>();
    boolean found = false;
    int max = 0;
    queue.add(s);
    while (!queue.isEmpty()) {
        String t = queue.remove();
        if (isValid(t)) {
            found = true;
            if (t.length() >= max && !result.contains(t)) {
                max = t.length();
                result.add(t);
            }
        }
        if (found) continue;
        for (int i = 0; i < t.length(); i ++) {
            String sub1 = t.substring(0, i);
            String sub2 = t.substring(i + 1, t.length());
            String temp = sub1.concat(sub2);
            if (!checked.contains(temp)) {
                queue.add(temp);
                checked.add(temp);
            }
        }
    }
    return result;
}
```

Remove Invalid Parentheses

```
public boolean isValid(String s) {
    int count = 0;
    for(int i = 0; i < s.length(); i ++){
        if (s.charAt(i) == '(') count ++;
        else if (s.charAt(i) == ')') {
            if (count == 0) return false;
            else count --;
        }
    }
    return (count == 0);
}
```

Homework

- Word Ladder
- Word Ladder II
- Surrounded Regions
- Number of Islands
- Remove Invalid Parentheses
- Find Bottom Tree Value
- Binary Tree Side View
- Find Largest Value in Each Tree Row