# Question

You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move consists of turning one wheel one slot.

The lock initially starts at '0000', a string representing the state of the 4 wheels.

You are given a list of deadends dead ends, meaning if the lock displays any of these codes, the wheels of the lock will stop turning and you will be unable to open it.

Given a target representing the value of the wheels that will unlock the lock, return the minimum total number of turns required to open the lock, or -1 if it is impossible.

**Example 1:**

**Input:** deadends = ["0201","0101","0102","1212","2002"], target = "0202"

**Output:** 6

**Explanation:**

A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202".

Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be invalid,

because the wheels of the lock become stuck after the display becomes the dead end "0102".

**Example 2:**

**Input:** deadends = ["8888"], target = "0009"

**Output:** 1

**Explanation:**

We can turn the last wheel in reverse to move from "0000" -> "0009".

**Example 3:**

**Input:** deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"

**Output:** -1

Explanation:

We can't reach the target without getting stuck.

**Example 4:**

**Input:** deadends = ["0000"], target = "8888"

**Output:** -1

**Constraints:**

* 1 <= deadends.length <= 500
* deadends[i].length == 4
* target.length == 4
* target **will not be** in the list deadends.
* target and deadends[i] consist of digits only.

# Solution

#### **Approach #1: Breadth-First Search [Accepted]**

**Intuition**

We can think of this problem as a shortest path problem on a graph: there are 10000 nodes (strings '0000' to '9999'), and there is an edge between two nodes if they differ in one digit, that digit differs by 1 (wrapping around, so '0' and '9' differ by 1), and if both nodes are not in deadends.

**Algorithm**

To solve a shortest path problem, we use a breadth-first search. The basic structure uses a Queue queue plus a Set seen that records if a node has ever been enqueued. This pushes all the work to the neighbors function - in our Python implementation, all the code after while queue: is template code, except for if node in dead: continue.

As for the neighbors function, for each position in the lock i = 0, 1, 2, 3, for each of the turns d = -1, 1, we determine the value of the lock after the i-th wheel has been turned in the direction d.

Care should be taken in our algorithm, as the graph does not have an edge unless both nodes are not in deadends. If our neighbors function checks only the target for being in deadends, we also need to check whether '0000' is in deadends at the beginning. In our implementation, we check at the visitor level so as to neatly handle this problem in all cases.

In Java, our implementation also inlines the neighbors function for convenience, and uses null inputs in the queue to represent a layer change. When the layer changes, we depth++ our global counter, and queue.peek() != null checks if there are still nodes enqueued.

|  |
| --- |
| class Solution {  public int openLock(String[] deadends, String target) {  Set<String> dead = new HashSet();  for (String d: deadends) dead.add(d);  Queue<String> queue = new LinkedList();  queue.offer("0000");  queue.offer(null);  Set<String> seen = new HashSet();  seen.add("0000");  int depth = 0;  while (!queue.isEmpty()) {  String node = queue.poll();  if (node == null) {  depth++;  if (queue.peek() != null)  queue.offer(null);  } else if (node.equals(target)) {  return depth;  } else if (!dead.contains(node)) {  for (int i = 0; i < 4; ++i) {  for (int d = -1; d <= 1; d += 2) {  int y = ((node.charAt(i) - '0') + d + 10) % 10;  String nei = node.substring(0, i) + ("" + y) + node.substring(i+1);  if (!seen.contains(nei)) {  seen.add(nei);  queue.offer(nei);  }  }  }  }  }  return -1;  }  } |

**Complexity Analysis**

