```
//
//  FENCHAO DU, HW8, GID=5
//  main.cpp
//  Created by Fenchao Du on 8/26/18.
//  Copyright © 2018 Fenchao Du. All rights reserved.
//
#include <iostream>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <queue>
using namespace std;

// HELPER function - return true if two strings have at most 1-char difference
bool iComp(const string & str1, const string & str2) {
    int flag = false;
    for(int i = 0; i < str1.size(); ++i) {
        if(str1[i] != str2[i]) {
            if(flag) { return false; }
            flag = true;
        }
    }
    return true;
}
// SOLUTION
// 1. If startWord and endWord have difference size, return false.
// 2. Reduce dictionary size by removing non-transitable words.
// 3. store nodes of current level in queue<string> graph
//      3.1 start radar from endWord. Only endWord in level 0
//      3.2 For level n, remove level n-1 words from graph.
//          store words that can be formed by replacing one char in level n-1
//      3.3 If 0 nodes in level n-1, stop searching level n!
//      3.4 We are looking for startWord in our radar
// 4. Because it's not weighted graph. Path to nodes is always shortest.
//    Erase words from dictionary to avoid duplicate/longer visit
// 5. store path information to map. key is source node, value is destination node.
//      5.1 endWord is our first destination node.
//      5.2 source node of level n-1, will be destination node of level n
//      5.3 The final destinaion for every node is endWord
// Performance:
// Assume word size = k, there are 'n' transitable words in dictionary.
// Transitable words have same size as start/end Word.
// Time
// string comparison is O(k)
// Suppose a1 words in level 1, a2 words in level 2,..., aj words in level j.
 a1+a2+...+aj = n
// Time = n+a1*(n-a1)+a2*(n-a1-a2)+a3*(n-a1-a2-a3)+...+aj*(n-a1-a2-a3-...-aj)
//      = n + n^2 - (a1*a1+a2*(a1+a2)+a3*(a1+a2+a3)+...+aj*(a1+a2+a3+...+aj))
// In fact, Dijkstra's algorithm, T(n) = O(|E|+|V|log|V|). In worst case, |E| =
 (n-1)*n/2
// T(n) = O(n^2); (less than n^2)
// Memory = O(n), in graph, we store at most all nodes
// in map, each node is a unique src node, meaning it cannot have multiple dst
// ↓↓↓↓↓ next page ↓↓↓↓↓
```

```cpp
bool isThereValidPath(unordered_set<string> dict, // make a copy
                      const string & startWord,
                      const string & endWord) {
    if (startWord.size() != endWord.size()) {
        cout << startWord << "-!->" << endWord << endl;
        return false;
    }

    for(auto & word : dict) {
        if(word.size() != startWord.size()) {
            dict.erase(word);
        }
    }

    queue<string> graph;
    unordered_map<string, string> mp;
    graph.push({endWord});
    dict.erase(endWord);
    dict.insert(startWord); // in case endWord == startWord, insert after erase
    while(!graph.empty()) {
        for(int i = graph.size(); i > 0; --i) {
            string dstWord = graph.front();
            graph.pop();
            for(auto & srcWord : dict) {
                if(iComp(dstWord, srcWord)) {
                    mp[srcWord] = dstWord;
                    if(srcWord == startWord) {
                        string word = startWord;
                        cout << startWord;
                        while(word != endWord) {
                            cout << "->" << mp[word];
                            word = mp[word];
                        }
                        cout << endl;
                        return true;
                    }
                    graph.push(srcWord);
                    dict.erase(srcWord);
                }
            }
        }
    }

    cout << startWord << "-!->" << endWord << endl;
    return false;
}

// TEST function
int main(int argc, const char * argv[]) {
    string startWord= "abcdef";
    string endWord = "kkkkkk";
    unordered_set<string> dict1({"kbcdef", "kkcdek", "kbcdek", "kkcokk", "kkcoek",
     "kkkokk"});
    isThereValidPath(dict1, startWord, endWord);
    unordered_set<string> dict2({"kbcdef", "kkcdek", "kbcdek", "kkcdkk"});
    isThereValidPath(dict2, startWord, endWord);
    isThereValidPath(dict2, startWord, startWord);

    return 0;
}
```