<https://leetcode.com/problems/word-ladder/>

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

* Every adjacent pair of words differs by a single letter.
* Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
* sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return *the* ***number of words*** *in the* ***shortest transformation sequence*** *from* beginWord *to* endWord*, or* 0 *if no such sequence exists.*

**Example 1:**

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

Output: 5

Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" -> cog", which is 5 words long.

**Example 2:**

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

Output: 0

Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the words in wordList are **unique**.

**Attempt 1: 2023-10-24**

**Solution 1: BFS level order traversal (10 min)**

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

Set<String> set = new HashSet<>(wordList);

if(!set.contains(endWord)) {

return 0;

}

Set<String> visited = new HashSet<>();

Queue<String> q = new LinkedList<>();

q.offer(beginWord);

visited.add(beginWord);

int step = 1;

while(!q.isEmpty()) {

int size = q.size();

for(int i = 0; i < size; i++) {

String cur = q.poll();

if(cur.equals(endWord)) {

return step;

}

for(int j = 0; j < cur.length(); j++) {

for(char c = 'a'; c <= 'z'; c++) {

char[] chars = cur.toCharArray();

if(chars[j] != c) {

chars[j] = c;

String next = new String(chars);

if(!visited.contains(next) && set.contains(next)) {

q.offer(next);

visited.add(next);

}

}

}

}

}

step++;

}

return 0;

}

}

Time Complexity: O(M^2 \* N), where M is size of dequeued word & N is size of our word list

Space Complexity:O(M \* N) where M is no. of character that we had in our string & N is the size of our wordList.

**Solution 2: Bi-direction BFS level order traversal (30 min)**

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

Set<String> set = new HashSet<>(wordList);

if(!set.contains(endWord)) {

return 0;

}

Set<String> visited = new HashSet<>();

Queue<String> q1 = new LinkedList<>();

Queue<String> q2 = new LinkedList<>();

q1.offer(beginWord);

q2.offer(endWord);

visited.add(beginWord);

visited.add(endWord);

int step = 1;

while(!q1.isEmpty() && !q2.isEmpty()) {

// Always check the smaller size queue and keep q1 as smaller one

// and q2 is by the way to keep the larger one

if(q1.size() > q2.size()) {

Queue<String> tmp = q1;

q1 = q2;

q2 = tmp;

}

Queue<String> nextQ = new LinkedList<>();

int size = q1.size();

for(int i = 0; i < size; i++) {

String cur = q1.poll();

for(int j = 0; j < cur.length(); j++) {

for(char c = 'a'; c <= 'z'; c++) {

char[] chars = cur.toCharArray();

if(chars[j] != c) {

chars[j] = c;

String next = new String(chars);

// If opponent queue contains next string then just need

// 1 more step to reach the target string

if(q2.contains(next)) {

return step + 1;

}

// Instead of use original wordList.contains(next), have

// to use set.contains(next) to improve speed

if(!visited.contains(next) && set.contains(next)) {

nextQ.offer(next);

visited.add(next);

}

}

}

}

}

// q1 already processed and update to nextQ

q1 = nextQ;

step++;

}

return 0;

}

}

Time Complexity: O(M^2 \* N), where M is size of dequeued word & N is size of our word list

Space Complexity:O(M \* N) where M is no. of character that we had in our string & N is the size of our wordList.

**Bi-directional BFS**

**Refer to**

<https://leetcode.com/problems/word-ladder/solutions/40711/Two-end-BFS-in-Java-31ms./>

// Solution: Bi-directional BFS

// https://leetcode.com/problems/word-ladder/discuss/40711/Two-end-BFS-in-Java-31ms.

// http://theoryofprogramming.com/2018/01/21/bidirectional-search/

/\*\*

http://www.noteanddata.com/leetcode-127-Word-Ladder-java-bidirectional-bfs-solution-note.html

题目 leetcode 127 Word Ladder

输入一个开始单词和结束单词， 然后在输入一个单词数组，

定义一个transform操作是从一个单词变换到另外一个单词， 然后这个变换是每次只能替换单词里面的一个字符到另外一个字符

问，最少需要经过多少个变换才可以从开始的单词到结束的单词， 返回的长度是变换列表的长度

解题思路分析

这个题目前面写了一个单方向的BFS, http://www.noteanddata.com/leetcode-127-Word-Ladder-java-bfs-solution-note.html，

但是运行时间比较慢， 更好的做法是双向BFS， 这次是第一次学写双向BFS

双向BFS的思路是

a. 从开始的节点走一层

b. 从结束的节点走一层

c. 交替上面两个步骤， 所以就是每次都走一层，看看走到这一层的时候是否会遇到对方的节点

双向BFS的步骤具体如下:

a. 把开始和结束的节点都加入到两个queue里面

b. 每次选择一个queue， 然后遍历一层, 如果遍历的下一层有对方节点，就退出

c. 选择另外一个queue， 然后遍历一层， 如果遍历的下一层有对方节点，就退出

d. 如果任何一个queue都空了，也就是没有办法继续向下一层，这时候如果还没有遇到可以结束的，就结束

java题解代码

这里用一个set来表示已经访问过的节点来去重

每次遍历的时候，用来一个小优化，就是每次从queue.size比较小的queue开始遍历，没有严格按照一个方向一层的方法来遍历，这样效果会更好一点

小细节要注意的是，双向BFS，每次还是要完整的做一层遍历，不能只做一个元素的遍历，否则就不对了

具体例子可以参考http://www.cppblog.com/Yuan/archive/2011/02/23/140553.html

"The idea behind bidirectional search is to run two simultaneous searches—one forward from

the initial state and the other backward from the goal—hoping that the two searches meet in

the middle. The motivation is that b^(d/2) + b^(d/2) is much less than b^d. b is branch factor, d is depth. "

----- section 3.4.6 in Artificial Intelligence - A modern approach by Stuart Russel and Peter Norvig