### **Word Ladder**

#### 理解问题

单词阶梯（word ladder），给定一本词典，一个初始单词start和一个目标单词target（他们长度相同），找到从start到target的最小链长度（如果存在），使得链中相邻单词只相差一个字符，链中的每个单词都是有效单词，即它存在于词典中。可以假设目标单词存在于词典中，并且所有词典单词的长度都是相同的。

Input : Dictionary={POON, PLEE, SAME, POlE, PLEA, PLIE, POIN}, start=TOON, target=PLEA

Output : 7

说明：TOON-POON-POIN-POIE-PLIE-PLEE-PLEA

Input : Dictionary ={ABCD, EBAD, EBCD, XYZA}, start=ABCV, target=EBAD

Output : 4

说明：ABCV-ABCD-EBCD-EBA

#### 算法分析

一开始想到首先构造一个无向图用来表示单词之间的可达性，然后从表示起点的节点开始对整个图进行BFS遍历，直到找到表示end的节点。

每次变化一个的这种类型的题优先考虑有BFS，每次控制变化其中一个字母，然后和dict中的比较，如果一样就加到树的下一层。count值由层数决定。这道题相当于树的最小深度。

#### 算法代码

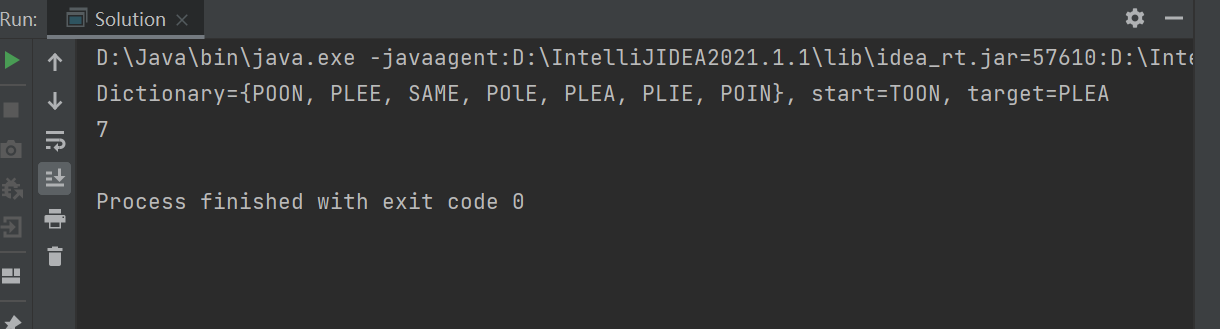
package Code;  
  
import java.util.\*;  
  
public class Solution {  
 public int ladderLength(String start,String end, Set<String> dict){  
 LinkedList<String> queue = new LinkedList<String>();  
 queue.add(start);  
  
 int count=1;  
 int cur=1;  
 int next=0;  
  
 while (!queue.isEmpty()){  
 String a = queue.poll();  
 if (dict.contains(a)){  
 dict.remove(a); //dict中有start的先删除  
 }  
 cur--;  
  
 char[] str=a.toCharArray();  
 for (int i = 0; i < str.length; i++) {  
 char tmp=str[i];  
 for (char j ='a'; j<='z';j++){  
 if (j==tmp)continue;  
 str[i]=j;  
 String b = new String(str); //注意不是toString 必须要先将str变为string才可以用下面的equals和contains判断  
 if (b.equals(end))return ++count; //要写在判断contains之前  
  
 if (dict.contains(b)){  
 queue.add(b);  
 next ++;  
 dict.remove(b); //remove  
 }  
 }  
 str[i]=tmp;//变化完记得还原，这样可以保证每次遍历只变了其中一个字母  
 }  
 if (cur==0){  
 cur=next;  
 next=0;  
 count++;//dict一层可能会有多个只变一个元素的 count应该在此处加而非dict.contains里面  
 }  
 }  
 return 0;  
 }  
  
  
 }

测试结果：

Input : Dictionary={POON, PLEE, SAME, POlE, PLEA, PLIE, POIN}, start=TOON, target=PLEA

Output : 7

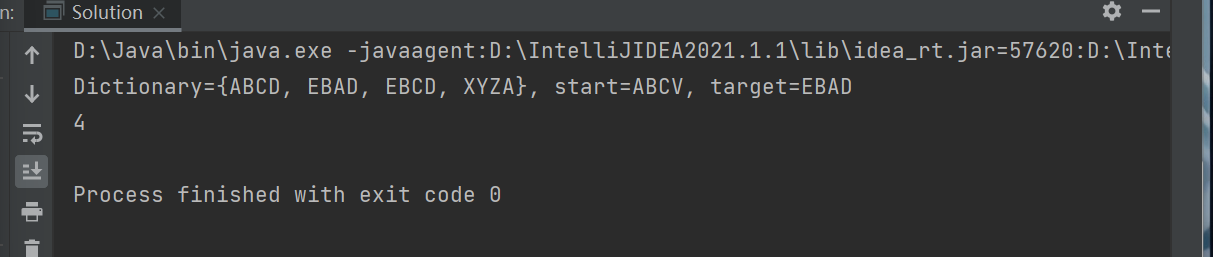
说明：TOON-POON-POIN-POIE-PLIE-PLEE-PLEA



Input : Dictionary ={ABCD, EBAD, EBCD, XYZA}, start=ABCV, target=EBAD

Output : 4

说明：ABCV-ABCD-EBCD-EBA



#### 算法优化

通过搜索资料发现可以在BFS遍历的时候去寻找当前单词可达的下一个单词。如果还是通过遍历所有的单词判断是否可达，则复杂度和上面一样，但实际上在上千个单词中，只有少数几个可以由当前单词一步到达，我们之前的比较浪费了很多时间在不可能的单词上。网上对该问题的解决无一例外都是按照下面的思路：将当前单词每一个字符替换成a~z的任意一个字符，然后判断是否在词典中出现。此时的复杂度是O(26\*word\_length)，当单词比较短时，这种方法的优势就体现出来了。

代码：

public int ladderLength(String start, String end, HashSet<String> dict) {  
 if (start == null || end == null || start.equals(end)  
 || start.length() != end.length())  
 return 0;  
  
 if (isOneWordDiff(start, end))  
 return 2;  
  
 Queue<String> queue=new LinkedList<String>();  
 HashMap<String,Integer> dist=new HashMap<String,Integer>();  
  
 queue.add(start);  
 dist.put(start, 1);  
  
 while(!queue.isEmpty())  
 {  
 String head=queue.poll();  
  
 int headDist=dist.get(head);  
 //从每一个位置开始替换成a~z  
 for(int i=0;i<head.length();i++)  
 {  
 for(char j='a';j<'z';j++)  
 {  
 if(head.charAt(i)==j) continue;  
  
 StringBuilder sb=new StringBuilder(head);  
 sb.setCharAt(i, j);  
  
 if(sb.toString().equals(end)) return headDist+1;  
  
 if(dict.contains(sb.toString())&&!dist.containsKey(sb.toString()))  
 {  
 queue.add(sb.toString());  
 dist.put(sb.toString(), headDist+1);  
 }  
 }  
 }  
 }  
  
 return 0;  
　　 }

### **二、**Cube Sort

#### 基本步骤

Cube sort 是一种对多维数组进行分治排序的算法，其基本思想是将待排序数组划分为多个立方体（cubes），然后分别对每个立方体进行排序，最后将这些有序的立方体合并成一个有序数组。下面是 CubeSort 算法的基本步骤：

划分立方体： 将待排序的数组划分为多个立方体，每个立方体包含固定数量的元素。立方体的数量可以根据实际情况调整，通常选择的大小与计算机内存大小和处理器缓存大小相关。

对每个立方体排序： 对每个立方体中的元素进行排序。可以选择任何一种排序算法来完成这个任务，通常选择效率较高的排序算法，比如快速排序或归并排序。

合并有序立方体： 将排序好的立方体按照一定规则合并，形成整体有序的数组。这一步可以使用归并操作来实现，将每个立方体的顶部元素依次放入一个优先队列（或堆）中，并逐步取出构建有序序列。

#### 时间复杂度

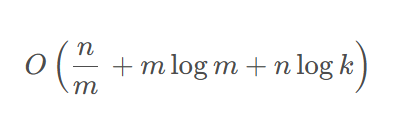
CubeSort 算法的时间复杂度取决于三个步骤：划分立方体、排序每个立方体以及合并有序立方体。

划分立方体： 划分立方体的时间复杂度取决于数组的大小和立方体的大小。假设数组大小为 n，立方体的大小为 m，则划分立方体的时间复杂度为 O(n/m)。

排序每个立方体： 对每个立方体进行排序的时间复杂度取决于所选的排序算法。通常情况下，可以选择效率较高的排序算法，比如快速排序或归并排序。假设所选的排序算法时间复杂度为 O(m log m)。

合并有序立方体： 合并有序立方体的时间复杂度取决于数组的大小和立方体的数量。假设立方体数量为 k，则合并有序立方体的时间复杂度为 O(n log k)。

综合考虑以上三个步骤，CubeSort 算法的总体时间复杂度为：



其中，n 表示数组的大小，m 表示立方体的大小，k 表示立方体的数量。立方体的大小和数量可以根据实际情况进行调整，以平衡算法的时间复杂度和空间复杂度。