**BFS——使用队列实现**

|  |  |  |  |
| --- | --- | --- | --- |
| 思路：找到满足条件的位置，从该位置的八个方向去查找下一个符合条件的位置。刚开始需要将第一个位置 **坐标** 加入队列，从该坐标向它的八个方向延伸，如果延伸到的位置满足条件，就将它的坐标加入队列，并且将路径长度加1保存到该满足条件的位置（此时为‘0’的值就改变为非0的值，从8个方向去找的时候，上个位置就不会再加入到队列，也就不会造成死循环），因为你有满足条件的下一个位置，所以队列中始终有下一个位置的坐标，每进行一次while循环，就弹出一个位置的坐标，直到队列为空，整个while循环查找结束。最后判断二维矩阵的最后一个位置是否为0即可，是0说明路径不能到达，非0说明有这样一条路径，且为最短路径。（如果能到达最后一个位置，其值一定为最短路径的原因，是因为有向斜角延伸这一步，向斜角延伸导致了它斜角这个位置保存的路径长度和向上下左右四个位置延伸保存的路劲长度一样）        思路：BFS——队列  将beginword加入队列，while循环中开始第一层的BFS。  开始每一层的BFS前需要判断该层有多少个元素，每一层的元素是等价的。  从beginword字符串开始，第一层只有一个元素，将该元素取出，依次将它的每个字母（比如这个字符串长度为len,先取第‘0’位置的字符）从’a’-’z’改变，将每次改变得到的字符串比较是否在字典中，如果在，就将该单词放入队列。第0位置的从a-z遍历完了，需要将该第0位置被改变的字符原来单词未改变时的状态，这样才能进行第1位置的a-z遍历，因此每次在进行a-z遍历之前，就需要将该位置的字符保存下来，从a-z遍历完了之后，又变回字符串原来的状态，这样就能保证每次只改变一个字符的要求，当0-len的字符都从a-z遍历完后，所有满足只改变一个字母条件的单词都会被加入队列。  紧接着进行第二层的BFS,因为该层的所有元素（这里代表一个个的字符串单词）是等价的，依次从队列中取出，再按照上面的步骤将字母从a-z改变，直到把该层的元素都取出遍历完。  而且每次从a-z产生的单词判断是否在字典中，如果在，判断是否与目标单词相同，相同则返回相应长度。因为每一层的所有元素等价，所以每次遍历完一层，路径长度才加1  首先需要判断endword是否存在于字典中，将字典放入HashSet即可，将beginword放入队列中，开始进行，还需要一个hashset记录使用过的单词，防止出现死循环   |  | | --- | | class Solution {      public int ladderLength(String beginWord, String endWord, List<String> wordList) {          //BFS一般用队列的形式实现          HashSet<String> set = new HashSet<>(wordList);          //判断边界条件，是否能直接返回结果          if(set.size() == 0 || !set.contains(endWord)){              return 0;          }          int step = 1;          int wordLen = beginWord.length();          //定义备忘录，记录已经访问过字符串          HashSet<String> visited = new HashSet<>();          //定义队列          Queue<String> queue = new LinkedList<>();          queue.offer(beginWord);          while(queue.size() != 0){              //获取当前队列中的元素个数（即该层有这么多个元素满足要求）              int currentSize = queue.size();              for(int i = 0; i < currentSize; i ++){                  //将要遍历的String保存起来                  String temp = queue.poll();                  char[] strCharArray = temp.toCharArray();                  for(int j = 0; j < wordLen; j ++){                      //保留第j个字母，从'a'-'z'遍历找字典中是否有改变第j位后的字符串，用于遍历完后复位第j个                      //字符，以便进行第j+1字符的遍历                      char originChar = strCharArray[j];                      for(char k = 'a'; k <= 'z'; k ++){                          if(strCharArray[j] == k){                              continue;                          }                          //固定将字符串的某一个字符(位置j)遍历改变，看是否在字典中                          strCharArray[j] = k;                          String charToString = String.valueOf(strCharArray);                          if(set.contains(charToString)){                              //判断是否到达位置                              if(charToString.equals(endWord)){                                  return step + 1;                              }                                if(!visited.contains(charToString)){                                  visited.add(charToString);                                  //把currentSize大小这一层满足要求的字符串加入到queue,为下一层的遍历做准备                                  queue.offer(charToString);                              }                          }                      }                      //第j位字母遍历完，需要将前面改变了的字符串恢复到原来的状态，接着从'a'-'z'遍历第j+1的字符                      strCharArray[j] = originChar;                  }              }              step ++;          }          return 0;      }  }  //时间复杂度O(26\*wordLen) |     BFS——队列   |  | | --- | | class Solution {      public int numSquares(int n) {          int res = 0;          Queue<Integer> queue = new LinkedList<>();          queue.add(n);          while(!queue.isEmpty()) {  int size = queue.size();//层序遍历大小                  for(int i = 0; i < size; i ++) {                  int temp = queue.remove();                  for(int j = 1; j < temp; j ++) {                      int tempNum = temp - j \* j;                      if(tempNum == 0) {                          return res + 1;                      }                      if(tempNum < 0) break;                      queue.add(tempNum);                  }              }  //一层遍历完之后才加1              res ++;          }          return n;      }  } | |  | |