COMP S265F Design and Analysis of Algorithms
Lab 7: Breadth-First Search

In this lab, we will apply Breadth-First Search (BFS) to solve a LeetCode problem "127. Word Ladder": https://leetcode.com/problems/word-ladder/

1. Word Ladder problem

A *transformation sequence* from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord -> ` s_1 `-> ` s_2 `-> ... -> ` s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for $1 \leq i \leq k$ is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- s_k `== 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.

```
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
```

The problem has given the following examples and constraints:

- **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:

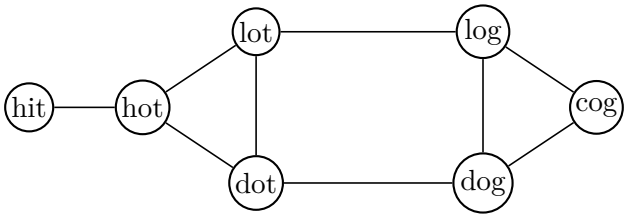
- `1 <= beginWord.length <= 10`
- `endWord.length == beginWord.length`
- `1 <= wordList.length <= 5000`
- `wordList[i].length == beginWord.length`
- `beginWord, endWord, and wordList[i]` consist of lowercase English letters.
- `beginWord != endWord`
- All the words in `wordList` are unique.

2. Problem formulation

We can formulate the problem as a graph problem:

- **Vertices**: `beginWord`, and words in `wordList`
- **Edges**: Two words w_1 and w_2 are connected by an edge if you can replace one character of w_1 to obtain w_2. The edges are undirected.

For Example 1, we have the following graph:



The problem is to find the shortest path distance from `beginWord "hit"` to `endWord "cog"`. We can use BFS to find this distance 4. Then, the answer to the word ladder problem is just $4 + 1 = 5$ (which is the number of vertices in the shortest path, instead of the number of edges).

Time complexity consideration. We use *adjacency list* to store the edges. One way to construct the adjacency list is to consider every pair of vertices and add an edge if they differ by only a single character. If `wordList` contains n words, the time complexity is $O(n^2)$. As n can be as large as 5000, this can be very slow.

An observation from the constraints is that every word is at most 10 characters long, and there are only 26 different characters at a position. Therefore, for each vertex, there are at most $26 \times 10 = 260$ words that differ by at most a single character. We can check whether this new word is one of vertices, and add an edge if this is the case. The time complexity is just $260n$.

On average, it takes $O(n)$ time to check membership in a list, but it only takes $O(1)$ time to check membership in a set or in a dictionary. Therefore, we can construct the graph, as follows:

lab07.py

```
from collections import defaultdict

class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        self.graph = defaultdict(list)
        wordList.insert(0, beginWord)
        self.numNodes = len(wordList)

        index = {}
        for i in range(len(wordList)):
            index[wordList[i]] = i

        if not endWord in index:
            return 0
        target = index[endWord]

        wordlen = len(beginWord)
        for i in range(self.numNodes):
            word = wordList[i]
            for j in range(wordlen):
                for c in range(26):
                    newword = word[:j] + chr(c+97) + word[j+1:]
                    if newword in index and newword != word:
                        self.graph[i].append(index[newword])
                        self.graph[index[newword]].append(i)
```

In the above code, `self.graph` is the adjacency list and `self.numNodes` is the number of vertices in the graph, where the vertices are labelled by the `wordList` index $0, 1, 2, ..., \texttt{numNodes} - 1$.

The dictionary `index` uses a word in `wordList` as the key, and keeps its index in `wordList` as value. Checking the membership of a word in `index` takes $O(1)$ time on average.

The ASCII code value of lowercase 'a' is 97. Note that the `chr(x)` function takes an integer ASCII code value `x` and returns the corresponding character in String format; while the `ord(y)` function does the reverse: takes a single character `y` and returns its integer ASCII code value.

Your task. Complete the above code by using BFS to find the number of vertices in the shortest path from `beginWord` (index 0) to `endWord` (index `target`).