Dictionary-based `wordNetwork`

- In our earlier attempt, we stored the word network in two lists: (a) a list of words and (b) a list of neighbors for each word in the word list.

- Now instead of using a neighbors list, we will use a dictionary for the neighbors:
  - `wordList`: the list of 5757 words
  - `neighborDict`: a dictionary whose keys are elements of `wordList` and the value of each key `w` is the list of neighbors of `w`.

- **Example**: So `neighborDict` will look like `{“aargh”: [], “abaca”: [“abaci”, “aback”], “abaci”: [“abaca”, “aback”], ...}`
# Loop to read words from the and to insert them in a list
wordList = []
for word in fin:
    newWord = word.strip("\n")
    wordList.append(newWord)
fin.close()

# Initialize Dictionary
neighborDict = {}
for w in wordList:
    neighborDict[w] = []

for i in range(len(wordList)):
    for j in range(i+1, len(wordList)):
        if areNeighbors(wordList[i], wordList[j]):
            neighborDict[wordList[i]].append(wordList[j])
            neighborDict[wordList[j]].append(wordList[i])
We are given a source node and a target node and asked to find a path through the word network from the source to the target.

We will design a network exploration algorithm; a specific version of this algorithm called breadth-first search yields shortest paths.

First we will focus on the problem of determining if there exists a source-target path. Later we will figure out how to store and report a source-target path.
Main Ideas

- Start at the *source*.

- At any point in the algorithm, there is a *current* node that the exploration is examining.

- From the current node, the algorithm picks one as-yet unexamined neighbor of the current node to examine next.

- This neighbors becomes the current node and the algorithm proceeds in this manner.
Main Ideas

- To avoid visiting the same set of nodes repeatedly – maybe infinitely – we will keep track of two sets of nodes.

- **Definition**: a node is said to be *processed* if it and all its neighbors have been examined.

- **Definition**: a node is said to be *reached* if it has been examined, but not all of its neighbors have been.

- We will maintain two sets of nodes: a *processed* set and a *reached* set.
In the “typical step” of the algorithm we will pick out a node from the reached set and process it.

**Pseudocode:**

- Pick an arbitrary node $w$ from the **reached** set
- For each neighbor of $w$:
  - If neighbor has not already been reached or processed then add the neighbor to the reached set.
- Add $w$ to **processed** set.
Stopping Conditions

- One of two things have to happen for the algorithm to stop:
  - If target is found, i.e., if target enters the reached set then we have detected a path from source to target.
  - If the reached set becomes empty, i.e., there is nothing left to explore, then there is no path from source to target.
# Explores the network of words D starting at the source word. If the
# exploration ends without reaching the target word, then there is no path
# between the source and the target. In this case the function returns False.
# Otherwise, there is a path and the function returns True.

def searchWordNetwork(source, target, D):
# Data structure initialization

## Initialization:

`processed` and `reached` are two dictionaries that will help in the exploration.

- **reached**: contains all words that have been reached, but yet not processed.
- **processed**: contains all words that have been reached and processed, i.e., their neighbors have also been explored.

The values of keys are not useful at this stage of the program and so we use 0 as dummy values.

```python
processed = {source:0}
reached = {}  
for e in D[source]:
    reached[e] = 0
```
Main loop

# Repeat until reached set becomes empty or target is reached
while reached:
    # Check if target is in reached; this would imply there is path from
    # source to target
    if target in reached:
        return True

    # Pick an arbitrary item in reached and process it
    item = reached.popitem()  # returns an arbitrary key-value pair as a tuple
    newWord = item[0]

    # Find all neighbors of this item and add new neighbors to reached
    processed[newWord] = 0
    for neighbor in D[newWord]:
        if neighbor not in reached and neighbor not in processed:
            reached[neighbor] = 0

return False
Questions

- How are we going to keep track of paths as we explore the network?

- What can we do to ensure that we are only exploring shortest paths?