Using Dictionaries for the Word Ladder Game
The three main actions one wants to perform on a data structure are: (i) search, (ii) insert, and (iii) delete.

On a dictionary $D$, one can perform these operations as follows:
- Search (boolean version): $\text{key in } D$ (key not in $D$)
- Search (regular version): $D[\text{key}]$
- Insert: $D[\text{key}] = \text{value}$
- Delete: del $D[\text{key}]$

All of these operations are extremely efficient.
Recall Dictionary Functions

1. `D.keys()` returns a list with all the keys in `D`
2. `D.values()` returns a list with all the values in `D`
3. `D.items()` returns a list of key-value pairs (as tuples)
4. `key in D, key not in D` evaluate to boolean values depending on whether key is in `D`
5. `D.pop(key)` removes the key-value pair corresponding to key and returns the value
6. `D.popitem()` removes and returns an arbitrary key-value pair from `D`
7. `D.update(uD)` updates `D` using the key-value pairs in `uD`
Data Structure for Word Network

- In the program we wrote last week, we used lists to construct a *word network* for the word ladders game.

- Recall:
  - `wordList` was a list to store words (in the order in which we read them from `words.dat`).
  - `neighborList` is a list of lists such that if a word `w` occurs in `wordList` in position `i` then all its neighbors are stored as a list, in position `i`, in `neighborList`.

- The problem with this data structure is that given a word `w`, it is costly to find its set of neighbors.
As before, we will use `wordList`.

In addition, we will use a dictionary called `neighborDict` for maintaining neighbors of words.

Each word $w$ in `wordList` will be a key in `neighborDict` with associated value being the list of neighbors of $w$.

**Example:** So `neighborDict` will look like

```
{"aargh": [], "abaca": ["abaci", "aback"], "abaci": ["abaca", "aback"],...}
```
Building a Dictionary-based word network

for i in range(len(wordList)):
    for j in range(i+1, len(wordList)):

        # Check if the two generated words are neighbors
        # if so append word2 to word1's neighbor list and word1 to
        # word2's neighbor list
        if areNeighbors(wordList[i], wordList[j]):
            neighborDict[wordList[i]].append(wordList[j])
            neighborDict[wordList[j]].append(wordList[i])
Experiment: Timing the two versions

1. Let us pick 10,000 words at random from wordList and place these words in a queryList.
2. For each word \( w \) in queryList, let us find the neighbors of \( w \).

- Let us time item (2) above, using the list-based word network and the dictionary-based word network.
- **Results:**
  - List-based: 0.459304094315 seconds
  - Dictionary-based: 0.00185298919678 seconds
Back to the word ladder game...

- Recall:
  - We are given a *source* word and a *target* word and asked to find a path, in the word network, from the source word to the target word.

- First, we will focus on determining if such a path exists.
- Later we will worry about recording the path.
• If we are at a particular word, we will then examine all neighbors of that word that have not already been examined.

• These new neighbors will be stored so that we can examine their neighbors later.
A word is said to have been *processed* if we have examined all its neighbors.

We will maintain two sets of words:

- **reached**: these are words that have been reached in the exploration, but not processed yet.

- **processed**: these are words that have been reached and have also been processed.
In the “typical step” of the algorithm we will pick out a word from the reached set and process it.

Pseudocode:
- Pick an arbitrary word $w$ 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.
def searchWordNetwork(source, target, D):
    
    # Initialization: processed and reached are two dictionaries that will help in the
    # exploration.
    # reached: contains all words that have been reached but 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.
    processed = {source:0}
    reached = {}
    for e in D[source]:
        reached[e] = 0

    # 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 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