127. Word Ladder

☼	12
👾 A	Imp
DSA 👻	BFS
topic 🦦	Graph
🦧 Desc	**Key Intuition: Pattern-Based Graph Construction** **Tip is :** Every adjacent pair of words differs by a single letter. **What the Pattern Actually Does:** The pattern acts as a **"meeting point"** for words that differ by exactly one character.

Tip is :

- Every adjacent pair of words differs by a single letter.

Problem Context

The goal is to find the shortest transformation sequence from `beginWord` to `endWord`, where each step changes exactly one letter and every intermediate word must exist in the `wordList`.

Key Intuition: Pattern-Based Graph Construction

The core insight is to treat this as a graph problem where words are connected if they differ by exactly one character. Instead of comparing every word pair (which would be $O(n^2)$), this solution uses an elegant pattern-matching approach:

Pattern Creation:

python

`pattern = word[:i] + "*" + word[i+1:]`

For each word, it creates patterns by replacing each character position with "*". For example:

- "hit" becomes: `["*it", "h*t", "hi*"]`
- "hot" becomes: `["*ot", "h*t", "ho*"]`

Graph Building:
Words that share the same pattern are one edit distance apart. In the example above, "hit" and "hot" both generate "h*t", so they're connected.

python

```
graph = collections.defaultdict(list)
# graph["h*t"] = ["hit", "hot", "hat", ...]
```

BFS Traversal

The algorithm then uses BFS to find the shortest path:

1. **Queue**: Stores `(current_word, transformation_count)`
2. **Visited**: Prevents revisiting words and infinite loops
3. **Level tracking**: Each time we move to a neighbor, we increment the transformation count

Why This Works

- **Correctness**: BFS guarantees we find the shortest path in an unweighted graph
- **Efficiency**: Pattern matching avoids comparing every word pair directly
- **Complete exploration**: The algorithm explores all possible one-character transformations systematically

The time complexity is $O(M^2 \times N)$ where M is word length and N is the number of words, which is much better than the naive $O(M \times N^2)$ approach of comparing every word pair.

The Problem Without Patterns

If we didn't use patterns to find all words that are one edit distance apart, we'd have to:

```
# Naive approach - compare every word with every other word
for word1 in wordList:
    for word2 in wordList:
        if isOneEditApart(word1, word2):  # O(M) comparison
            graph[word1].append(word2)
```

This is **O(N² × M)** - very expensive!

What the Pattern Actually Does

The pattern acts as a **"meeting point"** for words that differ by exactly one character.

Example Walkthrough:

Say we have words: `["hit", "hot", "hat", "lot"]`

Step 1: Generate patterns for each word

```
"hit" → ["*it", "h*t", "hi*"]
"hot" → ["*ot", "h*t", "ho*"]
"hat" → ["*at", "h*t", "ha*"]
"lot" → ["*ot", "l*t", "lo*"]
```

Step 2: Group words by shared patterns

```
graph = {
  "*it": ["hit"],
  "h*t": ["hit", "hot", "hat"],  # ← These 3 words share this pattern!
  "hi*": ["hit"],
  "*ot": ["hot", "lot"],         # ← These 2 words share this pattern!
  "ho*": ["hot"],
  "*at": ["hat"],
  "ha*": ["hat"],
  "l*t": ["lot"],
  "lo*": ["lot"]
}
```

The Key Insight

When we're at word "hit" and want to find its neighbors:

1. Generate patterns: `["*it", "h*t", "hi*"]`
2. Look up each pattern in the graph:

- `graph["*it"]` = `["hit"]` (just itself)
- `graph["h*t"]` = `["hit", "hot", "hat"]` (found neighbors!)
- `graph["hi*"]` = `["hit"]` (just itself)

So "hit" is connected to "hot" and "hat" because they all share the pattern "h*t".

Why This Works

The pattern `"h*t"` essentially means: *"any word that has 'h' as first letter, 't' as third letter, and anything as the second letter"*

Words sharing this pattern are **guaranteed** to be exactly one edit distance apart (they differ only in the '*' position).

The Efficiency Gain

Instead of:

- Comparing "hit" vs "hot" letter by letter ×
- Comparing "hit" vs "hat" letter by letter ×
- Comparing "hit" vs "lot" letter by letter ×

We do:

- Generate patterns for "hit" once ✓
- Instantly lookup all neighbors via shared patterns ✓

This reduces the complexity from **$O(N^2 \times M)$** to **$O(N \times M^2)$** - a massive improvement when N (number of words) is large!

The pattern is essentially a **pre-computed index** that tells us "here are all the words that differ from the current word by exactly one character at this specific position."

Complexity Analysis

- Time Complexity: $O(M^2 \times N)$, where M is the length of each word and N is the total number of words in the input word list.

- For each word in the word list, we iterate over its length to find all the intermediate words corresponding to it. Since the length of each word is *M* and we have *N* words, the total number of iterations the algorithm takes to create `all_combo_dict` is *M*×*N*. Additionally, forming each of the intermediate word takes *O*(*M*) time because of the substring operation used to create the new string. This adds up to a complexity of *O*($M^2 \times N$).
- Breadth first search in the worst case might go to each of the *N* words. For each word, we need to examine *M* possible intermediate words/combinations. Notice, we have used the substring operation to find each of the combination. Thus, *M* combinations take *O*(M^2) time. As a result, the time complexity of BFS traversal would also be *O*($M^2 \times N$).

Combining the above steps, the overall time complexity of this approach is *O*($M^2 \times N$).

- Space Complexity: *O*($M^2 \times N$)
  - Each word in the word list would have *M* intermediate combinations. To create the `all_combo_dict` dictionary we save an intermediate word as the key and its corresponding original words as the value. Note, for each of *M* intermediate words we save the original word of length *M*. This simply means, for every word we would need a space of M^2 to save all the transformations corresponding to it. Thus, `all_combo_dict` would need a total space of *O*($M^2 \times N$).
  - `Visited` dictionary would need a space of *O*(*M*×*N*) as each word is of length *M*.
  - Queue for BFS in worst case would need a space for all *O*(*N*) words and this would also result in a space complexity of *O*(*M*×*N*).

  Combining the above steps, the overall space complexity is *O*($M^2 \times N$) + *O*(*M*∗*N*) + *O*(*M*∗*N*) = *O*($M^2 \times N$) space.

Optimization:

We can definitely reduce the space complexity of this algorithm by storing the indices corresponding to each word instead of storing the word itself.