Topic Introduction

Today’s PrepLetter explores a classic and powerful pattern: **Breadth-First Search (BFS) for finding the shortest transformation path**.

BFS is an algorithm that explores all possible moves from a starting point, level by level, before moving deeper. Imagine standing at the center of a maze and sending out a wave in all directions. Each time the wave reaches cells further away, it visits all spots that are "distance k" from the start before exploring those at "distance k+1".

Definition:
BFS is a graph traversal algorithm that visits nodes by expanding outward from the starting node, layer by layer, using a queue data structure.

How it works:

- Start from the source node and mark it as visited.
- Add it to a queue.
- While the queue is not empty:
    - Remove the front node.
    - For each unvisited neighbor, mark as visited and add to the queue.

Why it’s useful in interviews:
BFS shines in *shortest path* problems on unweighted graphs. When interviewers ask you to find the minimum number of steps to get from one state to another (such as word transformations or mutations), BFS is often the best fit.

Quick Example (not one of today's problems):
Suppose you want to transform the word “cat” into “dog”, changing only one letter at a time, and each intermediate word must be in a dictionary. BFS would try all possible one-letter changes at each step, ensuring you find the shortest sequence if it exists.

Today’s three challenges are:

- Word Ladder
- Word Ladder II
- Minimum Genetic Mutation

These are grouped because each requires finding the shortest transformation path between two "states" (words or genes), where each transformation changes just one character and must be valid. Word Ladder I asks for the shortest path length, Word Ladder II for *all* shortest paths, and Minimum Genetic Mutation applies the same logic to gene strings.

Problem 1: Minimum Genetic Mutation

Minimum Genetic Mutation - LeetCode

Problem Statement (in my own words):
You’re given a starting gene string, an ending gene string, and a list of valid gene strings called the “bank”. Each gene is an 8-character string using ‘A’, ‘C’, ‘G’, ‘T’. In one mutation, you can change a single character to another valid one, but each

intermediate gene must be in the bank. What’s the minimum number of mutations needed to turn the start gene into the end gene? If impossible, return -1.

Example Input/Output:

Input:

start = "AACCGGTT"

end = "AACCGGTA"

bank = ["AACCGGTA"]

Output: 1

Explanation: Only one mutation needed (“AACCGGTT” -> “AACCGGTA”).

Thought Process:

- Each gene is a node.
- There’s an edge between two genes if they differ by exactly one character and both are in the bank.
- We want the shortest path from start to end.

Try sketching this on paper: draw nodes for each gene in the bank and draw edges between single-mutation neighbors.

Extra Test Case:

start = "AAAAACCC"

end = "AACCCCCC"

bank = ["AAAACCCC", "AAACCCCC", "AACCCCCC"]

Expected Output: 3

Brute-force Approach:

Try all possible mutation sequences. For each, check if it leads to the end. This is exponential in time, as there are 8 positions and 3 new choices per position (since each can be changed to one of three other letters).

- **Time Complexity:** O(4^N) in the worst case for N positions, very slow.

Optimal Approach:

Use BFS. Why? Because BFS finds the shortest path in unweighted graphs.

Step-by-Step Logic:

- If end gene is not in the bank, return -1 (impossible).
- Use a queue to store (currentGene, mutationCount).
- Use a set to track visited genes.
- For each gene, generate all possible one-letter mutations.
- If a mutation is in the bank and not visited, enqueue it with mutationCount+1.
- Stop when you reach the end gene.

Here’s the Python solution:

```python
from collections import deque

def minMutation(start: str, end: str, bank: list[str]) -> int:
    if end not in bank:
        return -1
    gene_length = len(start)
    bank_set = set(bank)
    visited = set([start])
    queue = deque([(start, 0)])
    possible_chars = ['A', 'C', 'G', 'T']

    while queue:
        current, mutations = queue.popleft()
        if current == end:
            return mutations
        for i in range(gene_length):
            for c in possible_chars:
                if c != current[i]:
                    mutated = current[:i] + c + current[i+1:]
                    if mutated in bank_set and mutated not in visited:
                        visited.add(mutated)
                        queue.append((mutated, mutations + 1))
    return -1
```

Time Complexity:

- Each gene can mutate at 8 positions into 3 new possibilities: up to 24 neighbors per gene.
- If B is the size of the bank: O(B *gene_length* 4) = O(B) in practice.

Space Complexity:

- Storing the bank and visited sets: O(B).

Explanation of the Code:

- We use a queue to do BFS.
- For each gene, we try changing each character to every other valid character.
- If the new gene is in the bank and not visited, we add it to the queue.
- We stop and return the mutation count when we reach the end gene.

Trace of Test Case:

Input:

start = "AACCGGTT"

end = "AAACGGTA"

bank = ["AACCGGTA", "AACCGCTA", "AAACGGTA"]

Step-by-step:

- Start with "AACCGGTT", mutation count 0.
- Try all mutations; "AACCGGTA" found in bank, add to queue.
- Next, from "AACCGGTA", try all mutations; "AAACGGTA" found, add to queue.
- Next, "AAACGGTA" matches the end gene, mutation count is 2. Return 2.

Try this test case yourself:

start = "AAAAACCC"

end = "AACCCCCC"

bank = ["AAAACCCC", "AAACCCCC", "AACCCCCC"]

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Word Ladder

[Word Ladder - LeetCode]

Problem Statement (rephrased):

Given a begin word, an end word, and a dictionary (word list), you can change only one letter at a time, and each intermediate word must be in the dictionary. What’s the length of the shortest transformation sequence from begin word to end word? If no such sequence exists, return 0. Only return the length, not the actual sequence.

How is this similar/different?

- Similar: Each state is a word; transformations are one-letter changes; find minimum steps.
- Different: Uses regular words, not genes; output is the *length* of the sequence.

Brute-force Approach:

Try every possible sequence of words. Exponential time.

Optimal Approach (BFS):

- If end word is not in the dictionary, return 0.
- Use a queue to store (word, steps_so_far).
- For each word, generate all possible one-letter changes.
- If the new word is in the dictionary and not visited, enqueue it.
- When end word is reached, return steps_so_far + 1.

Pseudocode:

```
function ladderLength(beginWord, endWord, wordList):
    wordSet = set(wordList)
    if endWord not in wordSet:
        return 0

    queue = [(beginWord, 1)]
    visited = set([beginWord])

    while queue is not empty:
        currentWord, steps = queue.pop_front()
        if currentWord == endWord:
            return steps
        for each position i in currentWord:
            for each letter 'a' to 'z':
                if letter != currentWord[i]:
                    nextWord = currentWord with i-th letter replaced by letter
                    if nextWord in wordSet and nextWord not in visited:
                        visited.add(nextWord)
                        queue.append((nextWord, steps + 1))
    return 0
```

Example Input/Output:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log","cog"]

Output: 5

Explanation: "hit" -> "hot" -> "dot" -> "dog" -> "cog"

Extra Test Case:

beginWord = "game"

endWord = "thee"

wordList = ["gane","gaze","gave","hate","have","thee"]

Expected Output: 0

(There’s no valid transformation.)

Trace:

Let's trace the main example:

- "hit" (start, step 1)
- Mutate: "hot" (step 2)
- "dot" (step 3)
- "dog" (step 4)
- "cog" (step 5), done.

Time Complexity:

- Each word: L letters, 25*L possible mutations.
- For N words: O(N *L* 26)

Space Complexity:

- O(N) for the word set and visited set.

Try dry-running this with:

beginWord = "lead"

endWord = "gold"

wordList = ["load", "goad", "gold"]

Problem 3: Word Ladder II

Word Ladder II - LeetCode

Problem Statement (rephrased):

Given a begin word, end word, and a word list, find *all* shortest transformation sequences from begin word to end word, changing one letter at a time and using only words from the list.

How is this different?

- Now, you must return every path that is of shortest length, not just the length itself.

Brute-force Approach:

Try every possible path. Too slow.

Optimal Approach:

Use BFS to find shortest paths, but also track how we reached each word so we can reconstruct all shortest sequences.

Step-by-Step Logic:

- Do BFS as before, but for each word, record all previous words that could reach it (predecessors).
- Stop BFS when you first reach the end word (all other shortest paths may be found within that BFS level).
- After BFS, use backtracking (DFS) from end word to begin word, reconstructing all paths using the predecessors map.

Pseudocode:

```
function findLadders(beginWord, endWord, wordList):
    wordSet = set(wordList)
    if endWord not in wordSet:
        return []

    layer = {beginWord: [[beginWord]]}

    while layer:
        new_layer = collections.defaultdict(list)
        for word in layer:
            if word == endWord:
                return layer[word]
            for i in range(len(word)):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    if c != word[i]:
                        newWord = word[:i] + c + word[i+1:]
                        if newWord in wordSet:
                                    new_layer[newWord] += [path + [newWord] for path in
layer[word]]
        wordSet -= set(new_layer.keys())
        layer = new_layer

    return []
```

Example Input/Output:

beginWord = "hit"

endWord = "cog"

wordList = ["hot","dot","dog","lot","log","cog"]

Output:

```
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

Extra Test Case:

beginWord = "a"

endWord = "c"

wordList = ["a", "b", "c"]

Expected Output: [["a","c"]]

Trace:

- Start with "hit".
- At each BFS level, expand possible words, tracking paths.
- When "cog" first appears in new_layer, collect all paths that reach it.
- Backtrack, reconstructing full sequences.

Time Complexity:

- BFS: Up to O(N *L* 26), but now we also store all shortest paths, so output size matters.
- Space: O(N * M), where N is the number of words, M is the average number of paths.

Try implementing this for:

beginWord = "red"

endWord = "tax"

wordList = ["red","ted","tex","tax"]

Summary and Next Steps

These three problems highlight a key interview pattern: **BFS for shortest transformation path**. Whether you’re transforming genes or words, thinking in terms of states and valid transitions helps you map the problem to a graph and use BFS efficiently.

Key Patterns & Insights:

- Use BFS for shortest path in unweighted graphs.
- When reconstructing all paths, track predecessors or full paths at each BFS level.
- Always check if the end state is reachable before starting.

Common Traps:

- Not marking nodes as visited early enough (can cause cycles).
- Accidentally allowing revisiting of nodes within the same BFS level (can cause exponential branching).
- For “all paths” versions, not stopping BFS at the first layer that reaches the end.

Action List:

- Solve all 3 problems on your own — even the one with code provided above.
- Try solving Problem 2 and 3 using a different technique, like bidirectional BFS.
- Explore other transformation path problems (e.g., Open the Lock, Sliding Puzzle).
- Compare your solution with others, especially for edge cases.

- If you get stuck, don’t worry! The key is steady practice and reflection.

Happy coding, and see you in the next PrepLetter!