*433 Minimum Genetic Mutation*

## Problem Description

In this problem, we are given a start gene string and an end gene string, each 8 characters long composed of the characters 'A', 'C', 'G', and 'T'. We also have a bank of valid gene mutations. A mutation is a change of a single character in the gene string, and a gene must be in the bank to be considered a valid gene mutation.

## Intuition

The intuition behind the solution is to treat each gene string as a node in a graph and each valid mutation as an edge connecting two nodes. We use BFS to explore the graph level by level, starting from the start gene and working towards the end gene. BFS is ideal for this scenario because it finds the shortest path (minimum mutations) from the start to the end node.

Here are the key steps of the BFS approach:

->Initialize a set from the bank to quickly check if a mutation is valid.

->Begin with a queue initialized with the start gene and a mutation count of 0.

->Use a dictionary to define mutation possibilities for each character.

->Dequeue an element and iterate over its characters, changing one character at a time according to the mutation possibilities.

->If a new mutation is valid (in the bank), enqueue it with a mutation count incremented by one.

->If we reach the end gene, return the mutation count.

->If the queue is exhausted without finding the end gene, return -1.

By using this BFS approach, we explore all possible gene mutations in the shortest number of steps, ensuring the minimum number of mutations needed to achieve the end gene, if possible.

## Implementation

```
s = set(bank)  # Convert bank into a set for O(1) access

q = deque([(start, 0)])  # Initialize the queue with a tuple of start gene and step counter

mp = {'A': 'TCG', 'T': 'ACG', 'C': 'ATG', 'G': 'ATC'}  # Mutation map



# BFS loop
```

```
while q:
  t, step = q.popleft()  # Pop the next gene string and current step count
  if t == end:  # End reached
    return step
  for i, v in enumerate(t):  # Iterate through each character in the string
    for j in mp[v]:  # Go through all possible mutations for the character
      next = t[:i] + j + t[i + 1:]  # Create the new mutated string
      if next in s:  # Check if the mutation is valid
        q.append((next, step + 1))  # Enqueue the new gene with incremented steps
        s.remove(next)  # Remove the visited gene from the set
return -1  # Return -1 if end can't be reached
```

## Time and Space Complexity

Considering that there are M gene sequences in the bank:

->In the worst case, the algorithm might have to visit all M sequences in the bank.

->For each sequence, we perform N character checks and 3 possible mutations.

->Each mutation check involves an O(1) set containment check and O(1) removal operation (since it's a set).

Thus, the total time complexity can be estimated as O(M * N * 3), which simplifies to O(M * N) as constants can be ignored in Big O notation.

## Space Complexity

The primary space-consuming structures are:

->The queue used for BFS which, in the worst case, might hold all M sequences before mutation at the same level.

->The set s, also holding the M gene sequences from the bank.

->The mp dictionary with a constant size of 4 (not dependent on N or M), therefore can be considered O(1).

So the space complexity is dominated by the queue and the set, both of which may hold up to M gene sequences. Hence, the space complexity is O(M).