**1. BFS나 DFS를 활용한 풀이**

- 알고리즘 개요

알파벳 순서를 추론하기 위해 BFS 기반의 위상 정렬(Kahn's Algorithm)을 사용했다.

- 알고리즘 설계

1. 그래프 생성

- 각 문자 간의 순서 관계를 표현하기 위해 방향 그래프를 만든다.

- 문자 간 관계는 단어 쌍을 비교하여 첫 번째로 다른 문자를 통해 유추한다.

- 각 문자의 진입 차수를 계산한다.

2. 위상 정렬

- 진입 차수가 0인 문자부터 BFS 방식으로 처리한다.

- 처리한 문자를 결과 순서에 추가하고, 연결된 문자의 진입 차수를 감소시킨다.

- 모든 문자를 처리할 때까지 반복한다.

3. 결과 검증

- 처리된 문자의 개수가 전체 문자 집합의 크기와 같아야 한다.

- 그렇지 않다면, 입력이 잘못된 것이거나 순서를 유추할 수 없는 경우이다.

- 구현 코드

from collections import defaultdict, deque

def find\_alphabet\_order\_bfs(words):

# 그래프 초기화

graph = defaultdict(set)

in\_degree = defaultdict(int)

# 그래프 구성

for i in range(len(words) - 1):

word1, word2 = words[i], words[i + 1]

min\_length = min(len(word1), len(word2))

for j in range(min\_length):

if word1[j] != word2[j]:

if word2[j] not in graph[word1[j]]:

graph[word1[j]].add(word2[j])

in\_degree[word2[j]] += 1

break

else:

if len(word1) > len(word2): # 접두어 관계 오류 방지

return ""

# 진입 차수가 0인 문자들로 BFS 시작

all\_chars = set("".join(words))

for char in all\_chars:

if char not in in\_degree:

in\_degree[char] = 0

queue = deque([char for char in in\_degree if in\_degree[char] == 0])

order = []

while queue:

char = queue.popleft()

order.append(char)

for neighbor in graph[char]:

in\_degree[neighbor] -= 1

if in\_degree[neighbor] == 0:

queue.append(neighbor)

# 처리된 문자 개수가 전체 문자 수와 다르면 오류 반환

if len(order) < len(all\_chars):

return ""

return "".join(order)

**2. 오일러 서킷을 활용한 풀이**

- 알고리즘 설계

1. 그래프 생성

- 각 단어를 간선으로, 단어의 시작 글자와 끝 글자를 정점으로 연결된 그래프를 만든다.

2. 오일러 서킷 조건 확인\*\*

- 각 정점의 진입 차수와 진출 차수를 계산한다.

- 모든 정점이 연결되어 있는지 확인한다.

3. 오일러 서킷 탐색

- Hierholzer’s Algorithm을 사용

- 구현 코드

from collections import defaultdict, deque

def find\_eulerian\_path(words):

# 그래프 초기화

graph = defaultdict(list)

in\_degree = defaultdict(int)

out\_degree = defaultdict(int)

# 그래프 구성

for word in words:

start, end = word[0], word[-1]

graph[start].append(word)

out\_degree[start] += 1

in\_degree[end] += 1

# 오일러 서킷 조건 확인

if not check\_eulerian\_circuit(in\_degree, out\_degree):

return "오일러 서킷을 구성할 수 없습니다."

# 오일러 서킷 탐색

circuit = []

stack = []

start\_node = next(iter(out\_degree)) # 시작점 선택 (아무 노드나 가능)

stack.append(start\_node)

while stack:

current = stack[-1]

if graph[current]:

# 방문하지 않은 간선이 있으면 스택에 추가

word = graph[current].pop()

next\_node = word[-1]

stack.append(next\_node)

else:

# 방문할 간선이 없으면 서킷에 추가

circuit.append(stack.pop())

circuit.reverse() # 역순으로 처리

return circuit[:-1] # 마지막 중복 노드 제거

def check\_eulerian\_circuit(in\_degree, out\_degree):

# 각 정점의 진입 차수와 진출 차수가 같아야 함

nodes = set(in\_degree.keys()).union(set(out\_degree.keys()))

for node in nodes:

if in\_degree[node] != out\_degree[node]:

return False

return True