1. 개요

이번 설계 과제에서는 POA #262 패턴 매칭 알고리즘을 구현해보고, 이를 통해 주어진 텍스트 내에서 특정 패턴을 찾는 과정을 살펴봅니다. 텍스트는 10글자 내외, 패턴은 3글자 내외로 설정하여 구현합니다. 알고리즘을 통해 실제 매칭되는 경우와 매칭되지 않는 경우를 모두 테스트합니다.

2. 알고리즘 분석 및 모듈화

패턴 매칭 알고리즘 POA #262는 주어진 텍스트 내에서 특정 패턴을 찾기 위한 일련의 절차를 따릅니다. 이를 더 작은 모듈로 나누면 다음과 같은 단계로 구성할 수 있습니다.

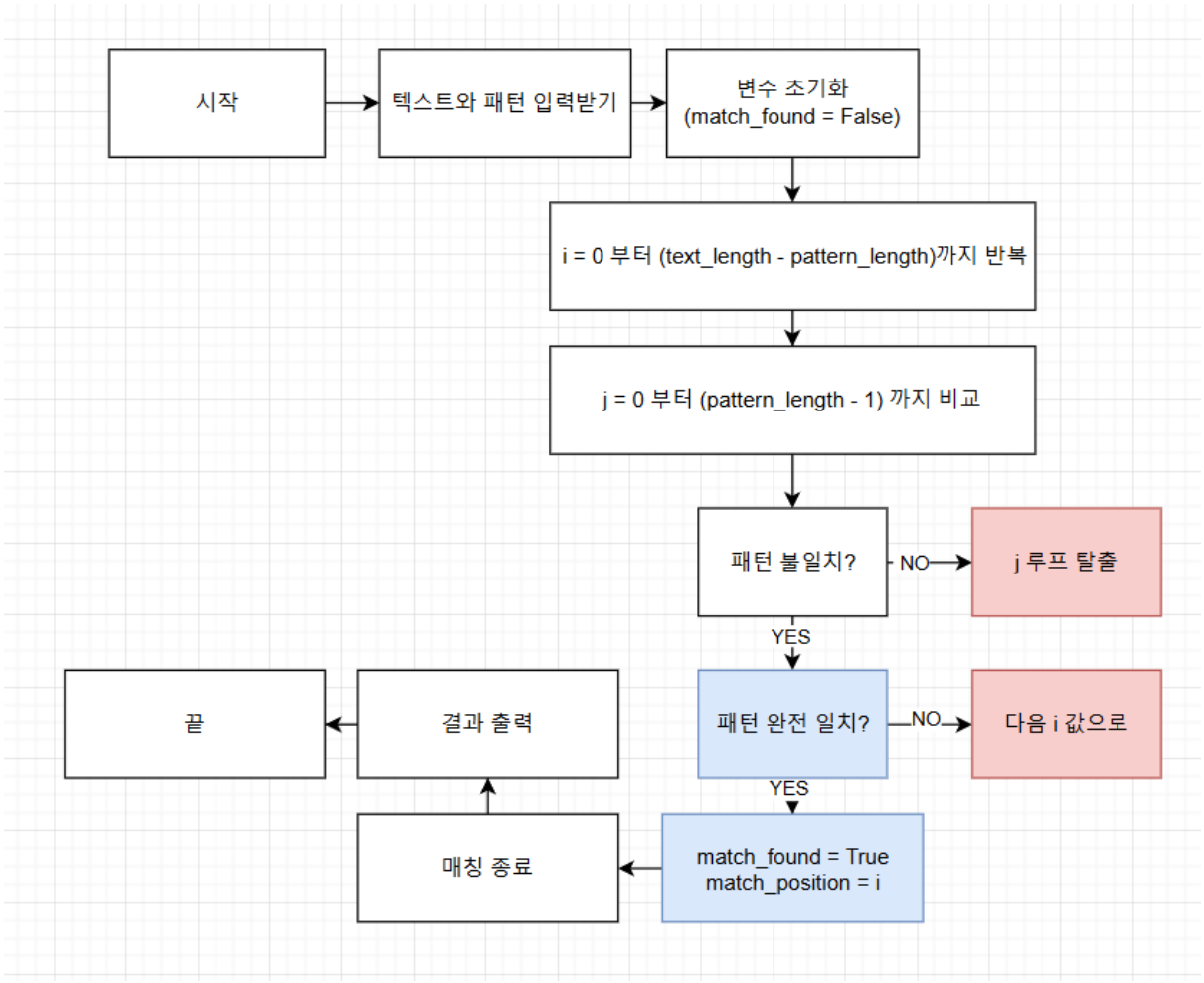
1. **입력 처리 모듈**
    - 텍스트와 패턴을 입력받음.
2. **패턴 매칭 모듈**
    - 텍스트를 순회하면서 패턴과의 일치 여부를 검사.
3. **결과 반환 모듈**
    - 매칭 여부와 위치를 반환하거나, 매칭되지 않음을 알림.

각 모듈의 변수 선언 및 용도는 다음과 같습니다.

- text: 검색할 전체 문자열. (예: "abcdefghij")
- pattern: 찾고자 하는 패턴 문자열. (예: "cde")
- text_length: 텍스트의 길이.
- pattern_length: 패턴의 길이.
- i, j: 반복문 인덱스 변수.
- match_found: 매칭 여부를 저장하는 불린 변수.
- match_position: 패턴이 매칭된 텍스트 내의 시작 인덱스.

3. 플로우차트(흐름도)

아래는 POA #262 알고리즘을 구현하기 위한 기본 플로우차트입니다.



main.py 436eq2ewg AI NEW PYTHON RUN

```python
def pattern_match(text, pattern):
    text_length = len(text)
    pattern_length = len(pattern)
    match_found = False
    match_position = -1

    # 텍스트 순회
    for i in range(text_length - pattern_length + 1):
        # 패턴과 텍스트 부분 비교
        j = 0
        while j < pattern_length and text[i + j] == pattern[j]:
            j += 1
        # 패턴이 완전히 일치하면
        if j == pattern_length:
            match_found = True
            match_position = i
            break  # 첫 번째 매칭 발견 시 종료

    return match_found, match_position

# 테스트 케이스
text = "abcdefghij"
pattern1 = "cde"  # 매칭되는 패턴
pattern2 = "xyz"  # 매칭되지 않는 패턴

# 패턴 매칭 실행 및 결과 출력
found1, position1 = pattern_match(text, pattern1)
found2, position2 = pattern_match(text, pattern2)

print(f"텍스트: {text}")
print(f"패턴1: {pattern1}, 매칭 여부: {found1}, 위치: {position1}")
print(f"패턴2: {pattern2}, 매칭 여부: {found2}, 위치: {position2}")
```

STDIN

Input for the program (Optional)

Output:

```
텍스트: abcdefghij
패턴1: cde, 매칭 여부: True, 위치: 2
패턴2: xyz, 매칭 여부: False, 위치: -1
```

5. 오류 지적 및 수정

구현 과정에서 발생할 수 있는 오류 예시

- **인덱스 범위 오류**: for i in range(text_length - pattern_length + 1)로 설정하여 텍스트의 끝 부분까지 비교하도록 함.
- **무한 루프 가능성**: 내부 while 루프에서 j 증감 조건을 잘못 설정하면 무한 루프가 발생할 수 있음. 이를 주의하며 구현.

현재 구현된 코드는 주요 오류 없이 정상적으로 동작하도록 설계되었습니다.

6. 성능 개선 방안

현재 알고리즘은 단순한 브루트 포스 방식으로, 모든 가능한 위치에서 패턴을 비교합니다. 텍스트와 패턴의 크기가 커질수록 성능 저하가 발생할 수 있습니다. 성능을 개선하기 위한 방안은 다음과 같습니다.

- **KMP 알고리즘 적용**: 패턴 전처리를 통해 불필요한 비교를 줄이는 방식.
- **보이어-무어 알고리즘 적용**: 뒤에서부터 비교하며 점프를 통해 비교 횟수를 줄이는 방식.
- **해시 기반 접근**: 롤링 해시를 이용하여 패턴과 텍스트 조각을 빠르게 비교.

POA #262가 특정 최적화 포인트를 가진다면, 해당 부분을 적용하여 성능 개선을 도모할 수 있습니다.

7. 개인 소감

이번 과제를 통해 기본적인 패턴 매칭 알고리즘을 이해하고, 이를 실제로 구현해보는 경험을 쌓을 수 있었습니다. 작은 텍스트와 패턴을 통해 알고리즘의 흐름을 명확하게 파악할 수 있었고, 모듈화와 변수 선언의 중요성을 다시 한 번 깨달았습니다. 특히 플로우차트를 그려보면서 각 단계별 처리 과정을 시각적으로 확인할 수 있었고, 이를 통해 디버깅과 개선점을 찾는 데 큰 도움이 되었습니다. 앞으로 더 복잡한 알고리즘을 구현할 때도 이번 경험을 바탕으로 단계별 설계와 최적화를 신경 써야겠다는 다짐을 하게 되었습니다.

7. 코드 첨부

```
def pattern_match(text, pattern):
    text_length = len(text)
    pattern_length = len(pattern)
    match_found = False
    match_position = -1

    # 텍스트 순회
    for i in range(text_length - pattern_length + 1):
        # 패턴과 텍스트 부분 비교
        j = 0
        while j < pattern_length and text[i + j] == pattern[j]:
            j += 1
        # 패턴이 완전히 일치하면
        if j == pattern_length:
            match_found = True
            match_position = i
            break  # 첫 번째 매칭 발견 시 종료

    return match_found, match_position

# 테스트 케이스
text = "abcdefghij"
pattern1 = "cde"  # 매칭되는 패턴
pattern2 = "xyz"  # 매칭되지 않는 패턴

# 패턴 매칭 실행 및 결과 출력
found1, position1 = pattern_match(text, pattern1)
found2, position2 = pattern_match(text, pattern2)

print(f"텍스트: {text}")
print(f"패턴1: {pattern1}, 매칭 여부: {found1}, 위치: {position1}")
print(f"패턴2: {pattern2}, 매칭 여부: {found2}, 위치: {position2}")
```