4주차 : KMP

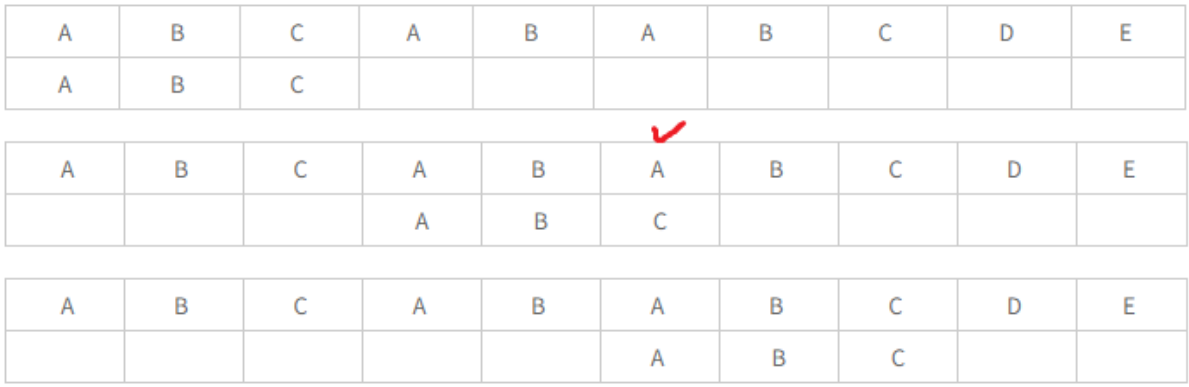
KMP 문자열 검색 알고리즘

```
커누스 모리스 프랫 알고리즘(Knuth-Morris-Pratt algorithm)
완전 탐색(Brute force)의 시간 복잡도 문제를 해결하기 위한 문자열 탐색 알고리즘
```

단순한 방법 (Brute Force)

“ABCABABCDE” 에서 “ABC”를 찾는다고 할 때

- 문자열의 맨 앞부터 맞는지 확인



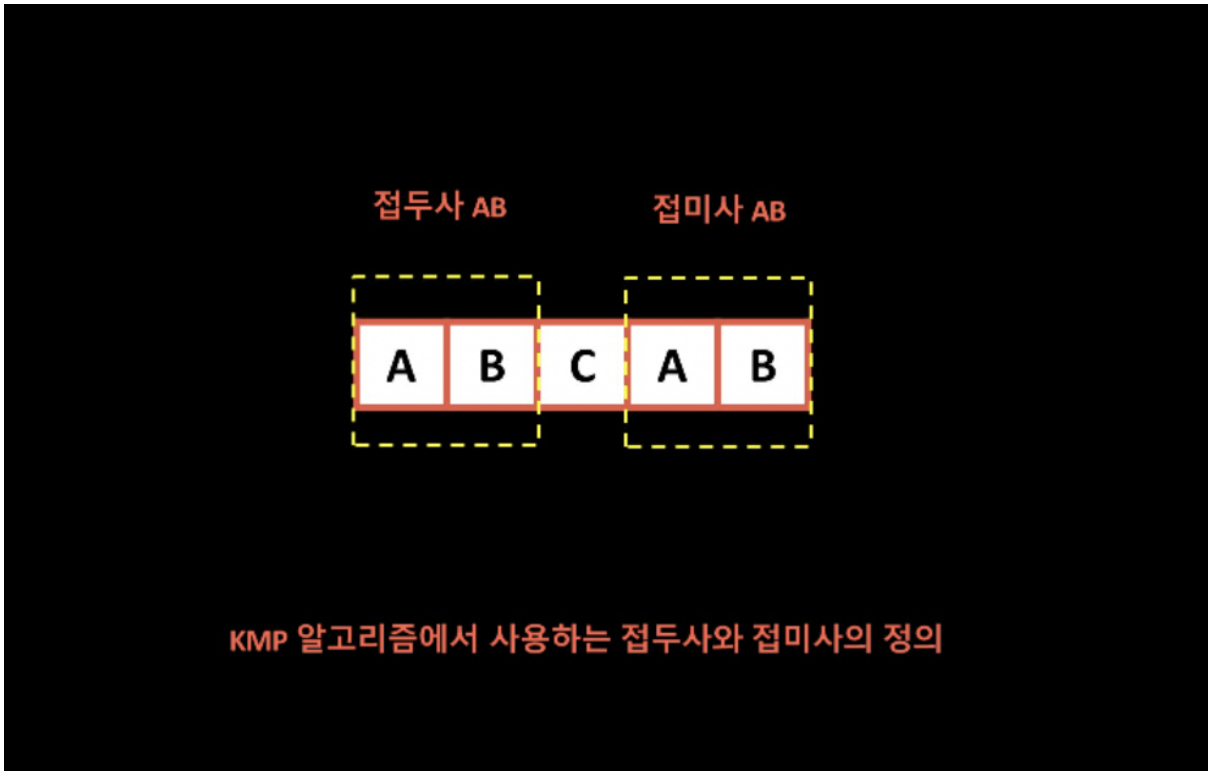
ABCABABCDE

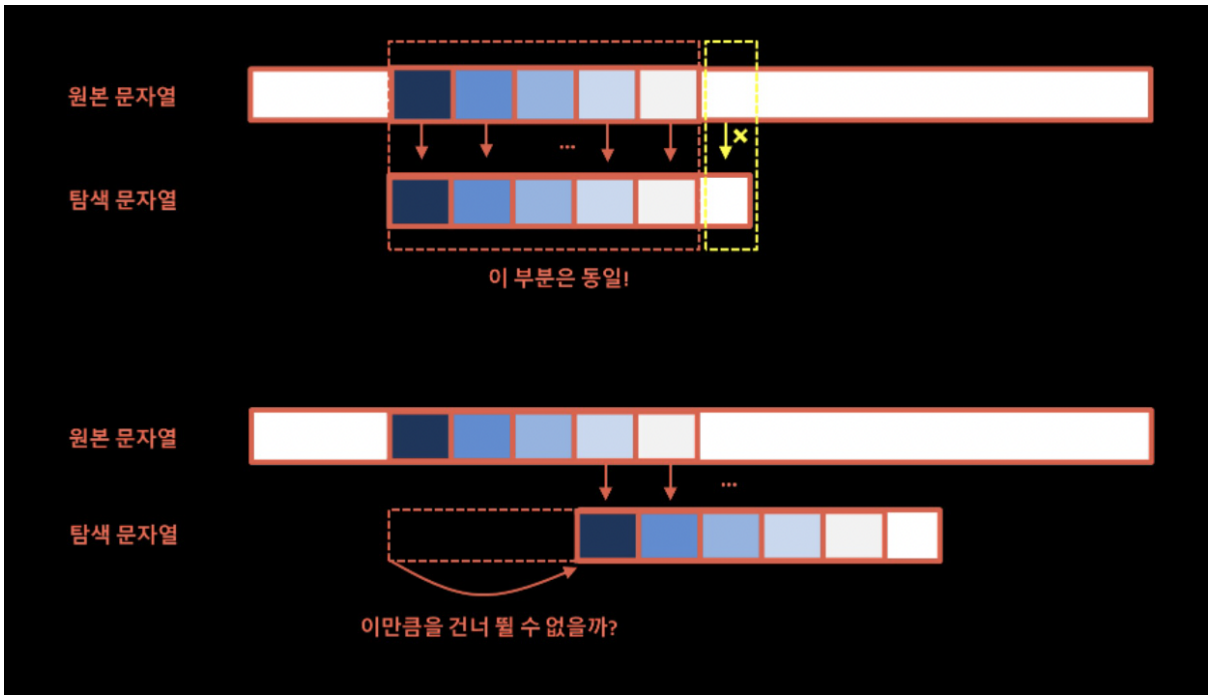
시간 복잡도는 텍스트의 길이(N), 패턴의 길이(M)이라고 할 때, 각 텍스트의 인덱스에 대해서 패턴이 일치하는지 비교하므로 **O(NM)**

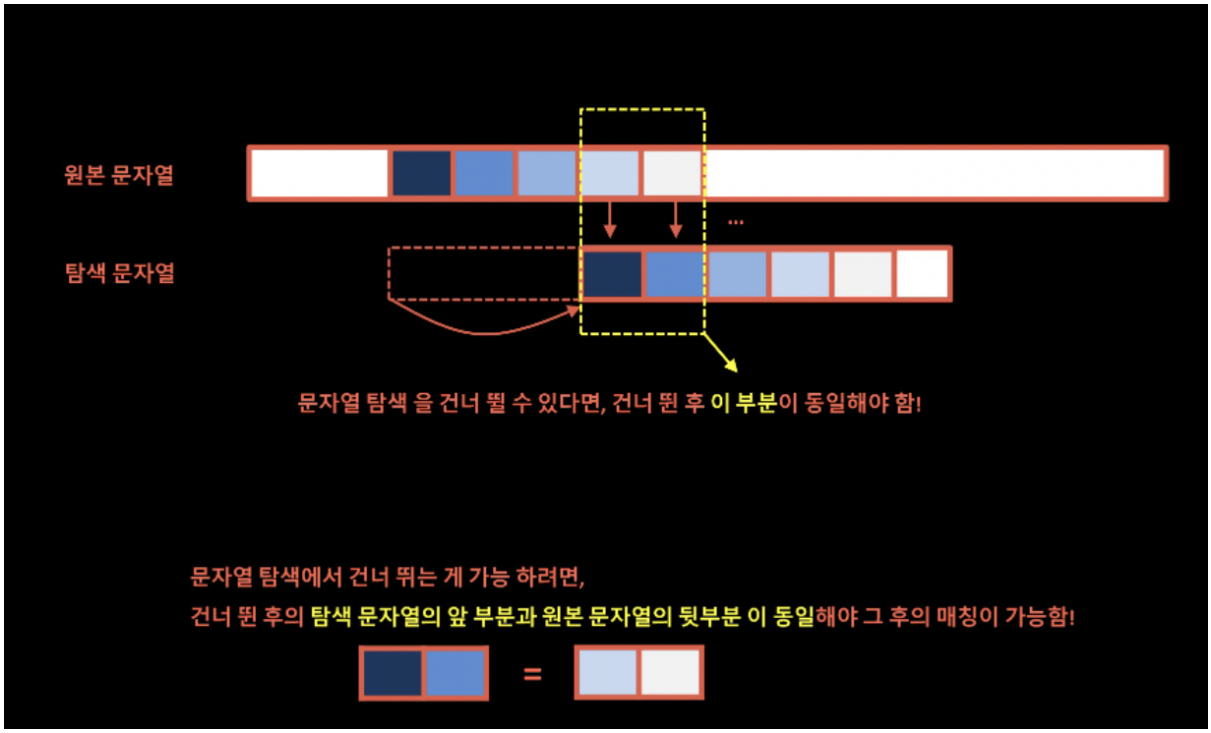
더 나은 방법을 찾아보자.

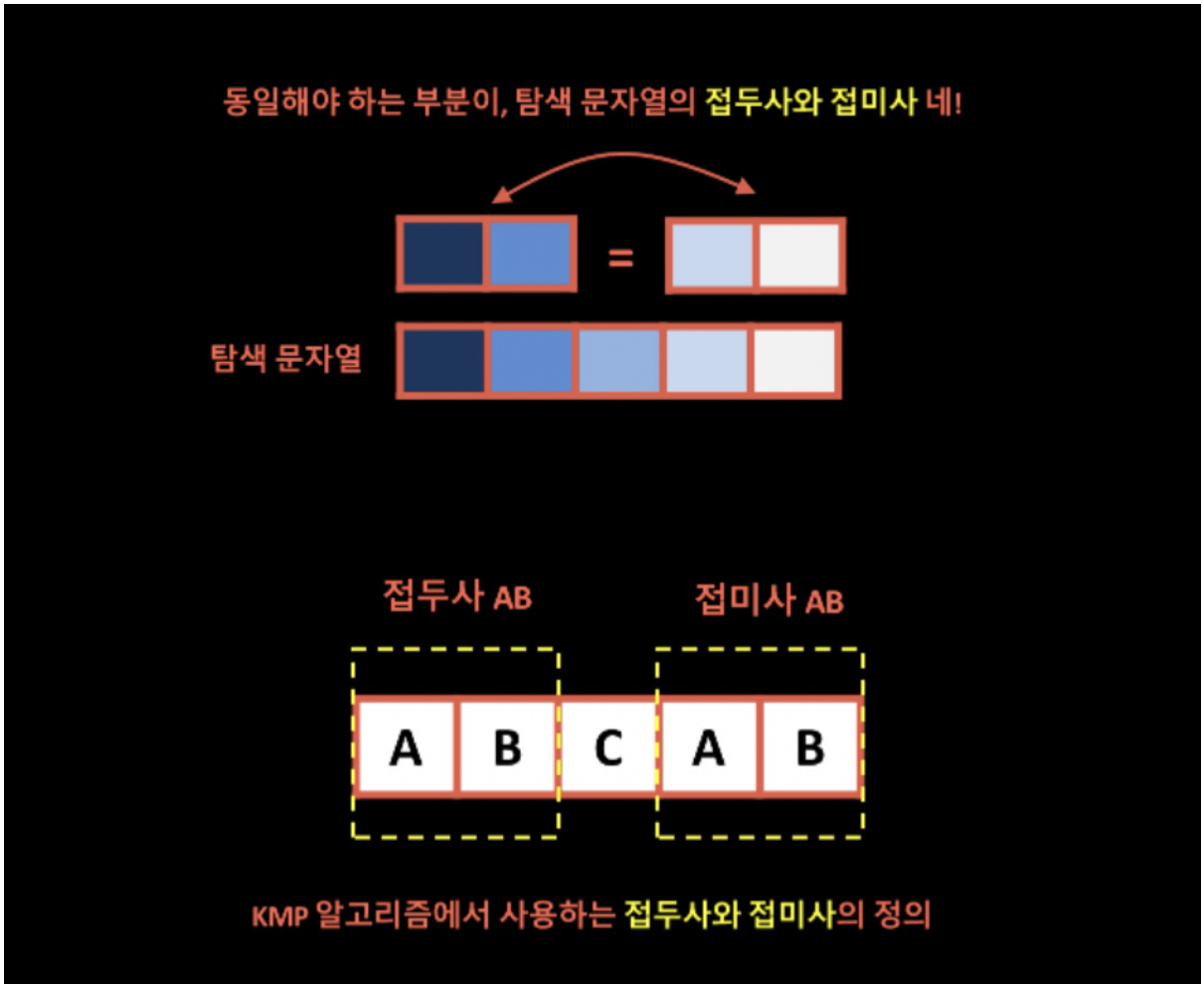
KMP 알고리즘

접두사와 **접미사**를 이용해 O(N+M)으로 문자열 탐색을 실행한다.
불일치가 발생하기 직전까지 같았던 부분은 다시 비교하지 않고 패턴 매칭(검색)을 진행하자!


접두사 AB
접미사 AB
A
B
C
A
B
KMP 알고리즘에서 사용하는 접두사와 접미사의 정의


원본 문자열
탐색 문자열
이 부분은 동일!
원본 문자열
탐색 문자열
이만큼을 건너 뛸 수 없을까?





source : https://injae-kim.github.io/dev/2020/07/23/all-about-kmp-algorithm.html

소스코드

```
# Pi array, LPS(Longest proper prefix which is suffix)
# prefix와 postfix가 같은 가장 긴 문자열 배열을 반환
def make_table(pattern):
    length = len(pattern)
    table = [0] * length
    j = 0

    for i in range(1, length):
        while j > 0 and pattern[i] != pattern[j]:
            j = table[j - 1]

        if pattern[i] == pattern[j]:
            j += 1
            table[i] = j

    return table


def kmp(source, pattern):
    table = make_table(pattern)
    source_length = len(source)
    pattern_length = len(pattern)

    j = 0
    # source 문자열의 인덱스를 하나씩 증가시키며 패턴과 비교.
    for i in range(source_length):
        # 소스와 패턴이 동일하지 않다면, LPS 배열만큼 이동하여 비교할 수 있다
        while j > 0 and source[i] != pattern[j]:
            j = table[j - 1]
        # 소스와 패턴이 동일하다면 다음 문자 비교
        if source[i] == pattern[j]:
            if j == pattern_length - 1:
                return 1
            else:
                j += 1

    return 0


# 인풋 입력부
source = input()
pattern = input()
print(kmp(source, pattern))
```

- while 문 예시

- "ABABABABBABABABABC"에서 패턴 "ABABABABC"를 찾는 상황

시간 복잡도

시간 복잡도는 텍스트의 길이(N), 패턴의 길이(M)이라고 할 때, 중복된 계산을 건너뛰고 있어 **O(N+M)**

기본 문제

https://www.acmicpc.net/problem/16916

추가 문제

https://www.acmicpc.net/problem/1701

참고

https://bowbowbow.tistory.com/6

https://injae-kim.github.io/dev/2020/07/23/all-about-kmp-algorithm.html