LPS & KMP Algorithm

0. Preview

⇒ 'bem'이 'benji'의 부분 문자열인지 알고 싶다면?

한 문자씩 대조해 가면서 알아보는 방법이 있는데, 시간복잡도가 최악의 경우 $O(N \times m)$ (bem의 길이 × benji 길이)

비효율적!!!

1. LPS (Longest prefix suffix)

1-1) 접두사(Prefix)와 접미사(Suffix)

⇒ LPS를 위해서 문자열 맨 앞에있는 접두사와 맨 뒤에 있는 접미사가 같은 경우를 먼저 찾아야한다.

ex) ABAAB ⇒ 이처럼 LPS는 서로 동일한 접두사와 접미사의 가장 긴 길이를 의미한다.

Prefix: AB / suffix: AB

1-2) 파이 배열

⇒ 위 문자열 'ABAAB'에서, 인덱스 별로 부분 문자열을 만들어 가며 살펴본다.

ex) [index] 인덱스 별로 검은 문자들의 부분 문자로 LPS를 구한다. (앞 ↔ 끝 비교)

		[LPS]
0	A B A A B	0
1	A B A A B	0
2	A B A A B	1
3	A B A A B	1
4	A B A A B	2

이 LPS 값을 배열로 정리하면, [0, 0, 1, 1, 2] 가 나오는데 위 배열을 **파이 배열** 이라고 한다!

또 다른 예제로, 'ABCABDAB'의 파이 배열을 구해 보자.

[index]	[pattern]	[LPS]
0	ABCABDAB	0
1	ABCABDAB	0
2	ABCABDAB	0
3	ABCABDAB	1
4	ABCABDAB	2
5	ABCABDAB	0
6	ABCABDAB	1
7	ABCABDAB	2

∴ LPS=[0,0,0,1,2,0,1,2]

★ 여기서 살펴볼 수 있는 점은, 이전 index의 LPS 값을 활용해서 현재 index의 LPS 값을 계산할 수 있다.

⇒ 엥..? 그렇다면 index 5번은 왜..? (이건 밑에 예제에서!)

[Example] 이때, 'pat=ABCABDAB'라 가정!

① 위에서 본 index==3 일 때, ABCABDAB 로 LPS 값이 1이었다. (동일한 접두사&접미사)

② 그 다음 index==4 일때, ABCABDAB 인데... 이전 LPS값에 1이므로 B가 추가되기전 접두사&접미사가 1개로 이미 일치한다. 즉, pat[0]==pat[3] 이다.
때문에, 이전 LPS에서 확인된 것은 생략하고 그 다음 문자만 비교하면 된다.
즉! pat[1]==pat[4] 만 확인하면된다. → 만약 같다면, 이전 LPS 값에 (+1)을 하면 된다.

③ 그렇다면, index 4와 5를 예시로 한번 살펴보자.
index==5 일 때, 0 1 2 3 4 5 ABCABDAB 가 된다. index==4 일 때 LPS가 2였다. (pat[2]==pat[4])

따라서 pat[2]==pat[5]를 검사해보는데.. 일치하지 않는다! 이럴때는.. pat[2]가 아닌 pat[0]이다.

LPS 배열에서 lps[최근 lps값-1]로 본다. 즉 lps[1]값 ⇒ pat[0]과 pat[5]를 비교한다.
↳ 이전 index 1까지의 LPS값

Q. 왜 이렇게 비교하는가? (알고 땡기는 이유!)

A. ABCABAAB 였다고 가정하면.. pat[0]==pat[5]가 될수도 있는 case가 존재하므로..!

2. KMP Algorithm

⇒ kmp는 knuth, Morris, Pratt 라는 세 명의 개발자 이름을 따온 명칭이다.
앞서 Pi 배열을 사용하여 불필요한 검사를 건너 뛴다.

2-1) 동작과정

원본 문자열과 부분 문자열을 비교한다.

A	B	C	D	A	B	C	D	A	B	E

A	B	C	D	A	B	E

① 운에 좋아 'ABCDAB' 까지는 일치했다.
그러나, 마지막 문자 E가 일치하지 않았다.

② 우리는 'ABCDAB' 까지, 즉 index가 5개지 일치했다는 사실을 알고 있다.
이 사실과 LPS값을 사용해서 불필요한 계산을 줄여보자.

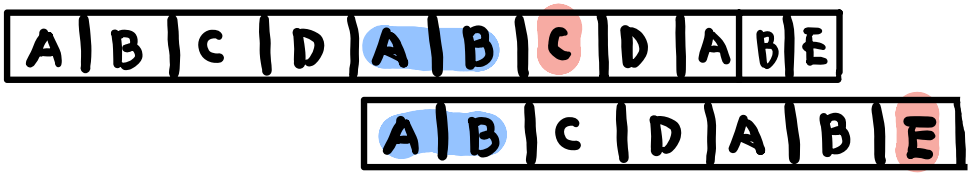
A	B	C	D	A	B	C	D	A	B	E

A	B	C	D	A	B	E

lps: 0 0 0 0 1 2

⇒ index 5에서의 LPS값은 2이다. 즉, 앞의 두 문자와 뒤의 두 문자가 일치한다는 의미이다.

③ 뒷 AB 자리에 앞 AB가 오도록 패턴을 움직여 볼 수 있다.



→ 새롭게 위치한 상태에서 첫 'AB'가 일치 한다는 것을 알고 있으므로 (KMP 원리) 뒤로가서 C 에서 검사를 새롭게 하면 된다!

∴ 시간 복잡도 O(M+N)

- M: 전체 문자열
- N: 부분 문자열