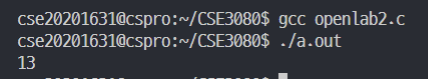
**오픈랩 과제 #2 : KMP Algorithm**

20201631 장유빈

**과제 실행 결과**



**KMP 알고리즘**

KMP 알고리즘은 패턴 매칭 알고리즘 중 하나이다. 즉, 주어진 패턴(pat : 주로 짧은 문자열)이 문자열(string : 패턴보다 긴 길이의 문자열)에 포함되어 있는지 확인하는 알고리즘이다. 단순한 알고리즘의 경우 매칭 지점의 위치를 1씩 증가시키면서 string의 가능한 모든 위치에서 비교하겠지만, 그보다 빠른 알고리즘이 KMP 알고리즘이다. KMP 알고리즘이 무엇인지는 다음 예시를 통해 확인해보자.

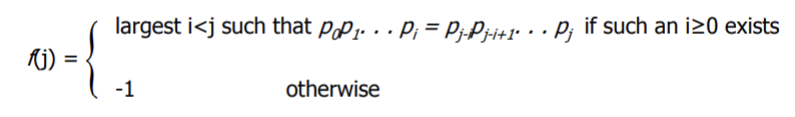
string[12] : ABCDABCDABEE

pat[7] : ABCDABE

다음과 같은 pat과 string이 있을 때 string이 pat을 포함하는지 확인하고자 한다. pat이 비교를 시작하는 string의 지점을 start라고 하자. 그렇다면 가장 먼저 start가 되는 인덱스는 0일 것이다. 즉, string[0]에서부터 비교를 시작하는 것이다. 그렇게 비교하게 되면 다음과 같이 string[6]에서 맞지 않는 문자를 발견하게 된다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| string | A | B | C | D | A | B | C | D | A | B | E | E |
| pat | A | B | C | D | A | B | E |  |  |  |  |  |

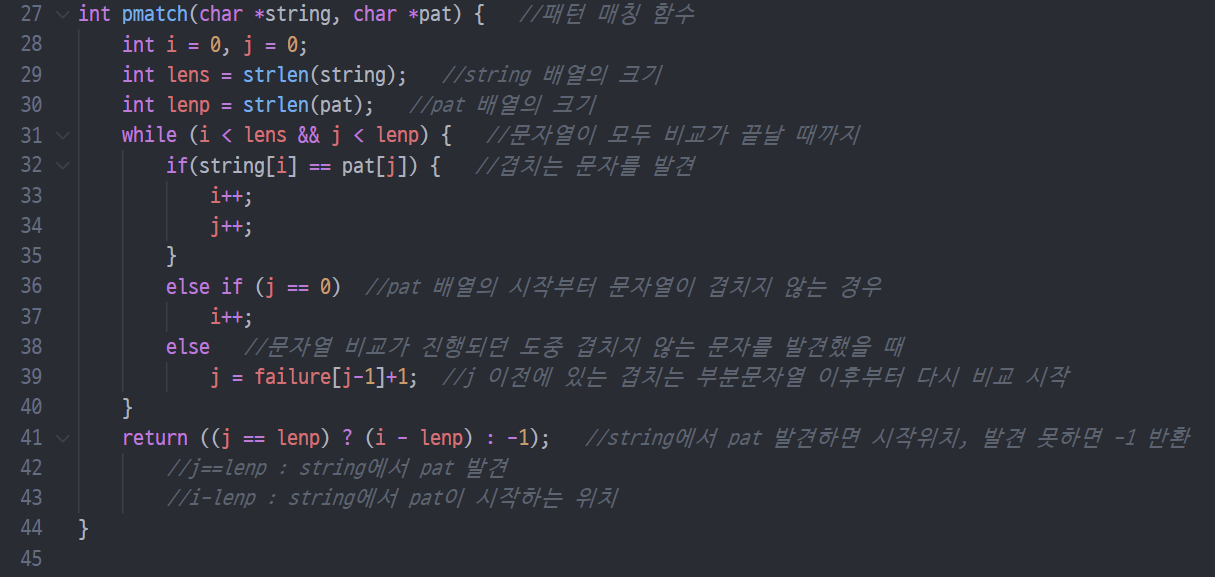
여기서 KMP 알고리즘과 단순한 알고리즘의 차이가 드러난다. 이 경우 단순한 알고리즘이라면 start가 1이 되어 string[1]인 B와 pat[0]인 A를 비교하게 될 것이다. 하지만 KMP 알고리즘의 경우 바로 다음으로 pat[0]이 일치하는 위치를 파악하여 start가 4가 될 것이다. 이를 가능하게 해주는 것이 failure 배열이다. failure 배열에서 failure[i] = f(i)라고 했을 때, 각 원소는 다음과 같은 방법으로 생성된다.

즉, pat에서 앞과 뒤에 같은 부분문자열이 존재한다면 그 길이에서 1을 뺀 값을, 존재하지 않는다면 –1을 대입한다. 위 예시의 pat을 통해 살펴보도록 하자.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 인덱스 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| pat | A | B | C | D | A | B | E |
| failure | -1 | -1 | -1 | -1 | 0 | 1 | -1 |

예시의 pat을 바탕으로 만들어진 failure 배열을 살펴보면, pat[3]까지는 pat[0]과 겹치는 문자열이 없으므로 failure 배열의 원소값은 모두 –1이다. pat[4]에서 처음으로 pat[0]과 같은 문자가 등장한다. 이때 앞과 뒤에 길이 1인 동일한 부분문자열이 등장했다고 볼 수 있으므로 길이에서 1을 뺀 0이 failure[4]의 값이 된다. 마찬가지로 pat[5]는 길이 2인 동일한 부분문자열인 AB가 앞과 뒤에 있으므로 길이 2에서 1을 뺀 1의 값이 failure[5]이다. 이후 pat[6]에서부터는 앞과 뒤에 겹치는 문자열이 없으므로 –1이 된다. 이 방식을 통해 컴퓨터는 패턴 매칭에 실패하면 어디서부터 다시 매칭을 시도해야 하는지 확인할 수 있게 된다. 이는 불필요한 비교 연산을 줄여주는 역할을 한다.

**코드**

설명은 주석을 통해 확인할 수 있다.

