5주차: KMP Algorithm

강사: 구건모

챕터 1: Naive String Searching Algorithm

우리가 흔히 쓰는 문자열 검색 알고리즘입니다!

std::string.find() / std::string.rfind() / strcmp / strstr 등등

Naive String Search

Naive String Search란 단순하고 직관적인 문자열 검색을 뜻한다.

문자열에 대하여 존재 여부 / 등장 횟수 / 등장 위치 등을 알아내기 위해 쓰인다.

Naive String Search

간단한 예시로 "ABCABABCDE" 문자열에서 "ABC" 가 몇번 등장하는지 세어보도록 하자.

Naive String Search

A	B	C	A	B	A	B	C	D	E
A	B	C							

Naive String Search

A	B	C	A	B	A	B	C	D	E
	A	B	C						

Naive String Search

A	B	C	A	B	A	B	C	D	E
		A	B	C	…				

Naive String Search

앞에서 봤던 예제처럼 Naive 하게 문자열 검색을 하게 되면

0번 인덱스에서 ABC가 존재하는지 확인
1번 인덱스에서 ABC가 존재하는지 확인
2번 인덱스에서 ABC가 존재하는지 확인
…
N – 1번 인덱스에서 ABC가 존재하는지 확인 (N: 검색의 주체가 되는 문자열의 길이)

이런 방식으로 문자열을 검색하게 된다

Naive String Search - 시간복잡도

앞에서 설명한 프로그램의 시간 복잡도를 계산해보도록 하자

본문의 문자열의 길이: N
검색하려는 단어의 길이: M
이라고 먼저 정의를 하자

1. 0 ~ N – 1 사이의 인덱스에서 매번 검색하려는 단어가 있는지 확인한다
2. 검색하려는 단어를 확인할 때 걸리는 횟수는 M이다.
3. 즉 N번 반복하며 반복할때마다 M번 확인한다.

알고리즘이 간단하니 다른 복잡한 증명도 없다.
O(NM) 이다.

Naive String Search - Pruning

Naive 한 풀이에서 Pruning 기법을 사용하여 커팅하면 실행시간을 줄일 수 있다.
말이 어려우니 간단한 예시를 보자.

"ABCDEFG" 라는 문자열에서 EFG 라는 단어를 찾는다고 하자.
그렇게 되면 0, 1, 2 .. 인덱스에서는 첫글자부터 맞지 않으니 비교할 필요가 없다.

"ABC" <-> "EFG" 를 비교할 때 "A"와 "E" 만 봐도 두 문자열은 다름을 알 수 있기 때문이다.
이렇게 비교를 하다 맞지 않는 문자가 나오면 틀린 문자열이라고 봐도 되니 실행시간을 줄일 수 있다.

Naive String Search – Pruning

이전 슬라이드에서 설명한 방법은 실행시간이 많이 줄어들 것이라고 생각되지만 예외가 있다.

"AAAAAAAAAAAAAAAAAAAAAAAAAB" 문자열에서 "AAAAAAAAAAAAB" 를 검색하는데 발생한다.

0번 인덱스에서 M – 1 번 비교를 하고 틀렸다고 판단을 할고
1번 인덱스에서 M – 1 번 비교를 하고 틀렸다고 판단을 하고
…
반복되기 때문에 최악의 상황에서는 O(NM)번 비교를 하기 때문이다.

따라서 Pruning 기법으로는 시간 복잡도를 줄일 수 없다.

Naive String Search

그렇다면 어떻게 해야 시간복잡도를 줄일 수 있을까??

챕터 2: KMP Algorithm

느린 문자열 검색 대신 빠른 문자열 검색을 위해 고안된 알고리즘이다.

KMP Algorithm

KMP Algorithm이란?
Knuth, Morris, Prett 세 사람이 만든 알고리즘으로 이름의 앞글자를 따 KMP 라는 이름이 되었다.

KMP 알고리즘이 어떤 방식으로 검색을 하길래 Naive한 알고리즘에서 최적화를 할 수 있을까??

KMP Algorithm

KMP Algorithm을 위해 알아야 할 것들이 있다. 아래 내용의 자세한 풀이는 뒷장에서 나온다.

1. Prefix (접두사)
2. Suffix (접미사)
3. Pi 배열 (Failure Function)

KMP Algorithm은 Prefix == Suffix를 통해 중복 검색을 막아 최적화하는 알고리즘이다.

말이 어렵지만 예제를 보면 직관적으로 이해할 수 있다.

KMP Algorithm

"ABCDABCDABEE" 라는 문자열에서 "ABCDABE" 를 검색해보도록 하자

KMP Algorithm

idx	0	1	2	3	4	5	6	7	8	9	10	11
T	A	B	C	D	A	B	C	D	A	B	E	E
P	A	B	C	D	A	B	E					

파란 글자는 매칭이 된 상태이고 6번에서 비교를 했을 때 문자열이 틀린 상태이다.

이 상태에서 Naive한 풀이로는 검색에 실패했으니 1번인덱스부터 다시 검색을 시작할 것이다.
하지만 1번인덱스에선 무조건 틀린다는 것을 알 수 있다.

이 때 KMP 알고리즘은 이 부분을 캐치하여 불필요한 검색을 막는다.

KMP Algorithm

idx	0	1	2	3	4	5	6	7	8	9	10	11
T	A	B	C	D	A	B	C	D	A	B	E	E
P					A	B	C	D	A	B	E	

0번인덱스에서 검색이 실패하고 KMP 알고리즘은 4번부터 다시 검색을 재개한다.
(사실 4번부터 검색하는게 아닌 6번부터 검색을 재개한다. 그 이유는 뒷장에 나온다.)

그냥 눈으로 보면 야 4번 인덱스부터 검색하면 되겠구나라고 알 수 있지만
이를 어떻게 알 수 있을지는 정확한 판단이 서지 않는다.

이를 천천히 풀어서 이해해보도록 하자.

KMP Algorithm

앞서 얘기했듯이 KMP 에서 최적화를 하는 방법은 Prefix == Suffix 라는 아이디어에서 시작된다.
간단한 예시를 생각해보도록 하자

"aaaaaaaaaaaaaaaaaab" 에서 "aaaab" 를 찾는다고 하자.
직관적으로 아 맨 오른쪽에 있구나 라고 생각 할 수 있지만, 조금 더 풀어서 생각해보도록 하자.

"aaaa...b 가 있다고 하면 우리는 먼저 0번 인덱스의 aaaa를 찾을 것이다.
그 다음엔 b가 없으니 맞는 단어가 아니고 다음 검색을 해야한다.

이 때 우리는 aaaa는 이미 맞은 문자열이기 때문에 1번인덱스부터 검색을 할 필요가 없다.
1번인덱스부터 3번 인덱스까지는 이미 aaa임이 확실하다.
그리고 우리가 검색하려는 문자열은 "aaaab" 이기 때문에 1번 인덱스부터 검색을 할 때
1, 2, 3번 인덱스는 맞았다고 생각할 수 있기 때문이다.
따라서 4번인덱스부터 다시 검색을 재개하며 3글자가 맞은 상태라고 판단할 수 있다.

이를 계속 반복하게된다면 "aaaab" 를 계속 검색하는게 아닌 "ab" 만 반복적으로 검색한다.

KMP Algorithm

위의 내용을 순차적으로 정리해보도록 하자.

1. 본문에 i번째 인덱스에서 단어를 검색한다. (초기 i값은 0이다)
2. 단어를 검색하는 도중 실패했을 때 (Failure Function)
2-1. 틀린 인덱스에서부터 다시 검색을 재개하되 단어에서 몇번째 문자부터 검색해야하는지 찾는다.
2-2. i 또는 i + 1 인덱스에서 몇글자가 맞았는지 설정해준 뒤 1번으로 돌아간다.
3. 검색에 성공했다면 이로써 kmp 알고리즘은 끝이 난다.

내용은 간단하지만 이를 어떻게 코드로 작성하는지는 떠오르지 않는다.
또 얼마나 빨라질지 감이 오지 않는다.

이 부분은 뒷장에서 나온다.

KMP Algorithm – prefix == suffix

KMP 알고리즘은 prefix == suffix 라는 아이디어가 메인이다. 그 이유가 이제 나온다.

앞에서 설명했듯이 검색에 실패했을 때를 생각해보자.
검색을 시작한 인덱스로 돌아가지 않고 검색을 실패한 인덱스부터 다시 검색을 시작한다.
그 이유는 검색에 일정부분 성공했다면 이전에 어떤 문자열이 있는지 알 수 있다. 라는 조건이 있다.

그렇다면 현재 위치 (i)에서 X개의 문자열이 맞은 상태에서 X + 1번째 문자가 틀렸다고 가정하면,
i – X ~ i 까지의 문자열은 검색하려는 단어의 앞 X글자임을 알 수 있다.
하지만 i – X 인덱스부터 검색을 하게 되면 검색에 실패함을 알 수 있다.

그렇다면 여기서 prefix == suffix 라는 아이디어가 적용된다.
i – X 인덱스에선 검색된 단어가 없으니 i에서 X개가 매칭된 상태는 단어 검색이 불가능함을 알수있다.
따라서 i인덱스에서 X미만의 최대 매칭 갯수를 다시 세어주는 것이다.

이 때 단어의 앞 X글자에 대하여 prefix 와 suffix 가 같은 부분 중 최대 길이를 찾게 되면,
i인덱스에서 X개 미만의 최대 매칭 갯수를 알 수 있다.

KMP Algorithm – pi 배열

Pi 배열이란 prefix == suffix 가 되는 최대의 길이를 담은 배열이라고 정의한다.
Failure Function 이라고도 한다. (실패했을 때 처리)

Pi[n] = 본문에서 단어가 n개 매칭된 상태에서 n미만의 prefix == suffix 가 되는 최대 길이를 뜻한다.
X개의 문자까지 매칭된 상태에서 매칭에 실패했을 때
실패한 인덱스에서 X 미만의 매칭되는 최대 길이를 구할 때 쓰인다.

예제를 살펴보자

KMP Algorithm – pi 배열

"ABAABAB" 라는 문자열을 검색한다고 하자.
따라서 "ABAABAB" 의 pi배열을 구하면 오른쪽 그림과 같다.

빨간색이 prefix 파란색이 suffix이다.

prefix == suffix 가 되는 최대의 길이를 pi 배열에 담았다.
이 pi 배열을 가지고 kmp 알고리즘을 구현해보도록 하자.

구현하는 방법은 뒤에서 다시 설명한다.

i	부분 문자열	pi[i]
0	A	0
1	AB	0
2	ABA	1
3	ABAA	1
4	ABAAB	2
5	ABAABA	3
6	ABAABAB	2

KMP Algorithm – kmp 구현

앞에서 설명했던대로 검색에 성공했을 때와,
검색에 실패했을 때, 두 분기로 나눠서 처리하면
간단하게 코드로 구현이 가능하다.

자세한건 주석 참조

```
void kmp() {
    // n: 본문의 길이 / m: 단어의 길이 //
    int n = strlen(str), m = strlen(word);
    // KMP 검색 시작 //
    // i: 인덱스 / mat: 단어에서 매칭된 갯수 //
    for (int i = 0, mat = 0; i < n; ++i) {
        // 현재 탐색중인 본문과 찾아야 할 다음 문자가 같을 때 //
        // 즉 검색에 성공했을 때 ( 문자열 검색 X 문자검색 O ) //
        if (str[i] == word[mat]) {
            // 모든 문자열을 검색에 성공했을 때 //
            if (++mat == m) {
                // 검색에 성공했을 때의 무언가 처리를 해준다 //

                // 현재 인덱스에서 mat 보다 작은 prefix == suffix 를 찾는다 //
                mat = pi[mat - 1];
            }
        } else if (mat != 0) { // 검색에 실패했을 때 //
            // 현재 인덱스부터 다시 검사한다 //
            --i;
            // 현재 인덱스에서 mat 보다 작은 prefix == suffix 를 찾는다 //
            mat = pi[mat - 1];
        }
        // mat이 0일 때에는 현재 인덱스에서 검색에 실패했을 때 //
        // 아무 의미가 없고 그냥 지나가면 되는 문자열이다 //
    }
}
```

KMP Algorithm – pi 배열 구하기

pi 배열을 구하기 위해서 prefix == suffix 인 부분을 찾아야 한다.
이 방법 역시 Naïve 하게 찾으면 $O(N^3)$ 으로 찾을 수 있다.

하지만 이 방법으로는 부족하다 더 빠른 방법을 찾아야 한다.

어떻게 빠른 방법으로 찾을 수 있을까??

KMP Algorithm - pi 배열 구하기

앞에서 kmp 알고리즘 방식대로 똑같이 최적화를 할 수 있다.

단어에서 단어를 찾는 느낌으로 접근하면 생각하기 쉽다.
똑같은 단어를 단어1 단어2 라고 생각해보자.
단어1의 인덱스를 i, 단어2의 인덱스를 j 라고 하자

이 때 단어1에 대해 단어2를 슬라이딩 시켜주면서 단어2의 prefix 가 단어1과 겹칠 때
Pi[i] = j 로 초기화 시켜줄 수 있다.
겹치지 않을경우 즉 실패했을 경우 Failure Function 을 통해 다시 검색을 재개하면 된다.

KMP Algorithm – pi 배열 구하기 구현

앞에서 설명했던대로 검색에 성공했을 때와,
검색에 실패했을 때, 두 분기로 나눠서 처리하면
간단하게 코드로 구현이 가능하다.

자세한건 주석 참조

```
void getPi() {
    // n: 단어의 길이 //
    int n = strlen(word);
    // pi 배열 구하기 시작 / pi배열은 모두 0으로 초기화한 상태 //
    // i: 인덱스 / mat: 단어에서 매칭된 갯수 //
    for (int i = 1, mat = 0; i < n; ++i) {
        // 검색 성공 //
        if (word[i] == word[mat]) {
            // pi 배열에 prefix == suffix 최대 길이 저장 //
            pi[i] = ++mat;
        } else if(mat != 0) { // 검색 실패 //
            // 현재 인덱스부터 다시 검사한다 //
            --i;
            mat = pi[mat - 1];
        }
    }
}
```

KMP Algorithm - 시간복잡도

KMP알고리즘의 시간복잡도를 계산하는 방법은 간단하다.

N: 주어진 문자열 (본문)
M: 검색할 단어 (단어)
|X|: 문자열 X의 길이라고 정의한다.

KMP알고리즘을 사용하는데 O(|N|)
pi배열을 구하는데 O(|M|)
의 시간이 든다.

따라서 KMP알고리즘의 전체 시간복잡도는 O(|N| + |M|) 이 된다.

BOJ 1786 찾기 - 풀이

```
#include <string.h>
#include <stdio.h>
#define MAX 1000001
char str[MAX];
char word[MAX];
int pi[MAX];
int res[MAX];
int rCnt;
void getPi() {
    // n: 단어의 길이 //
    int n = strlen(word);
    // pi 배열 구하기 시작 / pi배열은 모두 0으로 초기화한 상태 //
    // i: 인덱스 / mat: 단어에서 매칭된 갯수 //
    for (int i = 1, mat = 0; i < n; ++i) {
        // 검색 성공 //
        if (word[i] == word[mat]) {
            // pi 배열에 prefix == suffix 최대 길이 저장 //
            pi[i] = ++mat;
        } else if(mat != 0) { // 검색 실패 //
            // 현재 인덱스부터 다시 검사한다 //
            --i;
            mat = pi[mat - 1];
        }
    }
}
```

```
void kmp() {
    // n: 본문의 길이 / m: 단어의 길이 //
    int n = strlen(str), m = strlen(word);
    // KMP 검색 시작 //
    // i: 인덱스 / mat: 단어에서 매칭된 갯수 //
    for (int i = 0, mat = 0; i < n; ++i) {
        // 현재 탐색중인 본문과 찾아야 할 다음 문자가 같을 때 //
        // 즉 검색에 성공했을 때 ( 문자열 검색 X 문자검색 O ) //
        if (str[i] == word[mat]) {
            // 모든 문자열을 검색에 성공했을 때 //
            if (++mat == m) {
                // 검색에 성공했을 때의 무언가 처리를 해준다 //
                res[rCnt++] = i - m + 1;
                // 현재 인덱스에서 mat 보다 작은 prefix == suffix 를 찾는다 //
                mat = pi[mat - 1];
            }
        } else if (mat != 0) { // 검색에 실패했을 때 //
            // 현재 인덱스부터 다시 검사한다 //
            --i;
            // 현재 인덱스에서 mat 보다 작은 prefix == suffix 를 찾는다 //
            mat = pi[mat - 1];
        }
        // mat이 0일 때에는 현재 인덱스에서 검색에 실패했을 때 //
        // 아무 의미가 없고 그냥 지나가면 되는 문자열이다 //
    }
}
```

BOJ 1786 - 풀이

```
int main() {
    char in[1 << 15], out[1 << 18];
    setvbuf(stdin, in, _IOFBF, sizeof(in));
    setvbuf(stdout, out, _IOFBF, sizeof(out));
    gets(str);
    gets(word);
    getPi();
    kmp();
    printf("%d\n", rCnt);
    for (int i = 0; i < rCnt; ++i) {
        printf("%d ", res[i] + 1);
    }
    return 0;
}
```

감사합니다!

끝