5️⃣

알고리즘 : 문자열 매칭

문자열 매칭(String maching)

- 입력

A[1...n]: 텍스트 문자열
P[1...m]: 패턴 문자열
단, n>>m // n이 m에 비해 훨씬 크다고 가정한다.

- 수행작업

텍스트 문자열 A[1...n]이 패턴 문자열 P[1...m]을 포함하는지를 알아본다. 또한 포함한다면 어떤 위치인지를 알아낸다.

- 예

A = “aababacccc”, P = “aba” 일 때 매칭이 일어난 위치(A의 인덱스)는 2와 4이다. (매칭이 두 군데 서 일어나며, 매칭 시작 인덱스를 출력)

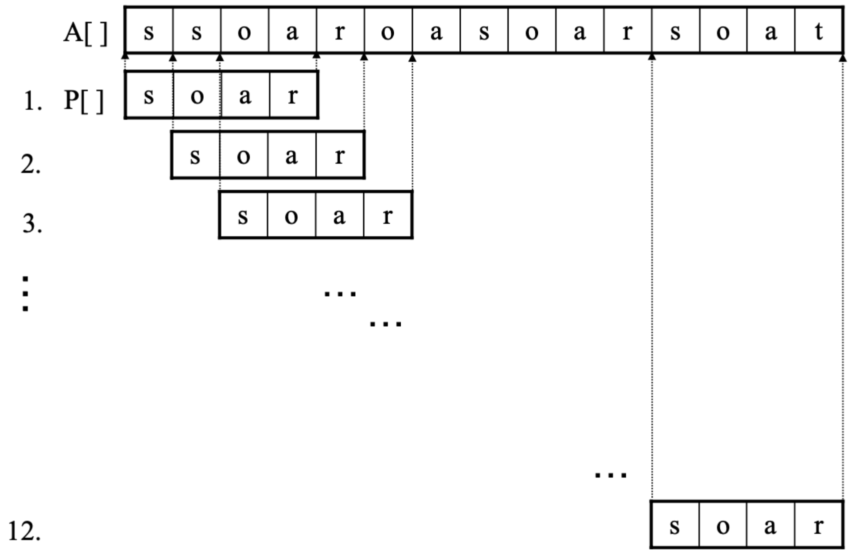
원시적인(naive) 매칭 방법

```
naiveMatching(A[], P[]) { // n: 배열 A의 길이, m: 배열 P의 길이
  for i ← 1 to n-m+1{   // n-m+1 => O(n)번 반복
    if (P[1...m] = A[i...i+m-1]) then   // O(m)번 반복
      A[i] 자리에서 매칭이 일어났음을 알린다;
  }
}
```

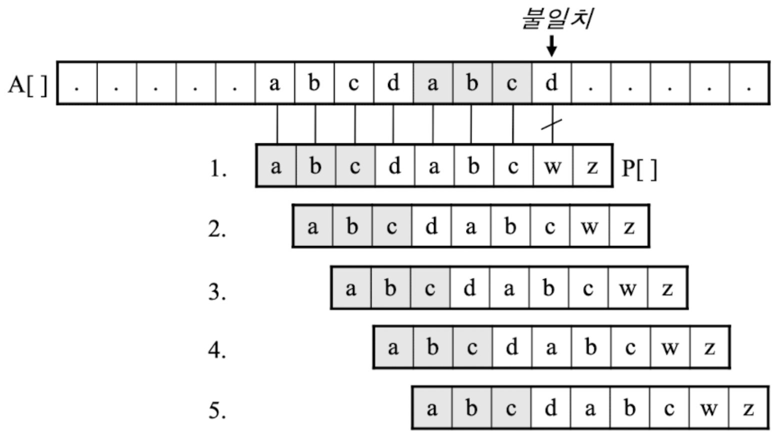
=> 수행시간 : O(mn)시간이 소요된다.

원시적인 매칭의 작동 원리

처음부터 하나하나 다 비교한다.



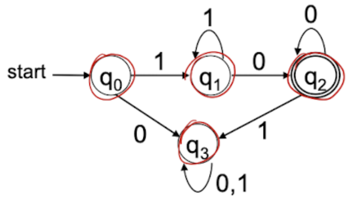
원사적인 매칭이 비효율적인 이유



- 앞선 매칭 과정에서 얻은 정보를 전혀 이용하지 않으므로 비효율적 : 1에서 불일치로 중단하지만, P의 앞부분 abc가 A의 두번째 abc와 일치한다는 사실을 활용할 수 있다면 2, 3, 4의 비교는 불필요하다.

오토마타를 이용한 매칭 방법

- 예)



$\delta(q_0, 0) = q_3$
$\delta(q_0, 1) = q_1$

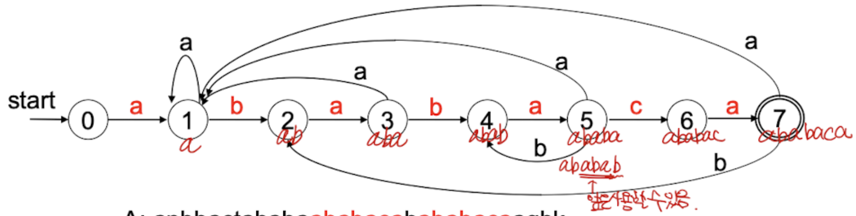
Q 상태집합 $= \{q_0, q_1, q_2, q_3\}$
q_0 시작상태
F 목표상태 (최종 상태)들의 집합 $= \{q_2\}$
Σ 입력 알파벳 $= \{0, 1\}$
δ 상태 전이 함수 $Q \times \Sigma \to Q$

매칭 단계 1 : 전처리 작업 (preprocessing)

찾고자 하는 패턴에 대해 오토마타를 정의해야 한다.

예)
P[1...m] ➔ 하나의 오토마타
매칭이 어디까지 진행되었는가를 상태로 표현하고, 어떤 경우 어떤 상태로 전이하는가를 정의한다.

패턴 ababaca에 대한 오토마타



A: anbbactababaababacabababacaagbk…
P: ababaca

오토마타 (Q, q_0, F, ∑, δ)
Q = {0, 1, 2, 3, 4, 5, 6, 7} (m+1개)
q_0 = 0
F = {7}
∑ = {a, b, c, d, ... , z}
δ : 다음 슬라이드

오토마타에 입력 알파벳이 표시되지 않은 경우 매칭 실패로서, 시작 상태로 전이한다고 본다.

매칭 단계 2 : 매칭 (matching)

시작 상태에서 시작하여 텍스트 문자열의 문자를 하나씩 읽어 그 문자에 맞게 상태 전이한다.
최종 상태에 이르면 매칭된 것이다. 이어서 텍스트 문자열을 계속해서 읽고 상태 전이도 계속한다.

```
FA-Matcher(A, δ, f ) {     // f : 목표 상태
  q ← 0;          // 0: 시작 상태
  for i ← 1 to n {    // n: 배열 A[ ]의 길이
    q ← δ(q, A[i]);
    if (q = f) then A[i-m+1]에서 매칭이 발생했음을 알린다.;
  }
}
```

=> 수행시간은 Θ(n)시간이 소요된다. 상태 전이 함수 구성 시간은 가장 효율적이라고 볼 때 Θ(|∑|m) 시간이 소요된다.

라빈-카프 알고리즘

문자열 패턴을 수치로 바꾸어(수치화) 문자열의 비교를 수치 비교로 대신한다.

수치화 : 가능한 문자 집합의 크기에 따라 진수가 결정된다. |∑|를 d라고 한다.

- 수치화 예

∑ = {a, b, c, d, e, f, g, h, i, j}
|∑|=10
a, b, c, ... , j 를 각각 0, 1, 2, ... , 9 에 대응시킨다.
문자열 “cad”를 수치화하면 *2102+0*101+3*100 = 203

해결해야 하는 문제점

수치화 작업의 부담이 되고, 수치가 커져 오버플로우 발생 가능성이 있다.

=> 이 두가지 문제점을 해결하여 문자열 매칭을 수행한다.

- 수치화 작업의 부담

P[1...m]에 대응하는 수치 p의 계산
p = ((...((P[1]*d)+P[2])d + ...)d+P[m -1])d+P[m]*
예) “edabc” ➔ p = 454 + 353 + 052 + 151 + 2 = ((((4*5)+3)*5+0)*5+1)*5+2

A[i...i+m-1]에 대응하는 수치 ai 의 계산
ai = ((...((A[i]*d)+A[i+1])d + ...)d+A[i+m-2])d+A[i+m-1]*
예) “dcedabcc” ➔ a1 = 354 + 253 + 452 + 351 + 0 = ((((3*5)+2)*5+4)*5+3)*5+0

문제점 : ai 계산에 Θ(m)의 시간이 든다. ai를 일일이 계산하는 경우 A[1...n] 전체에 대한 비교는 Θ(mn)이 소요된다. ➔ 원시적인 매칭에비해 나은게 없다.

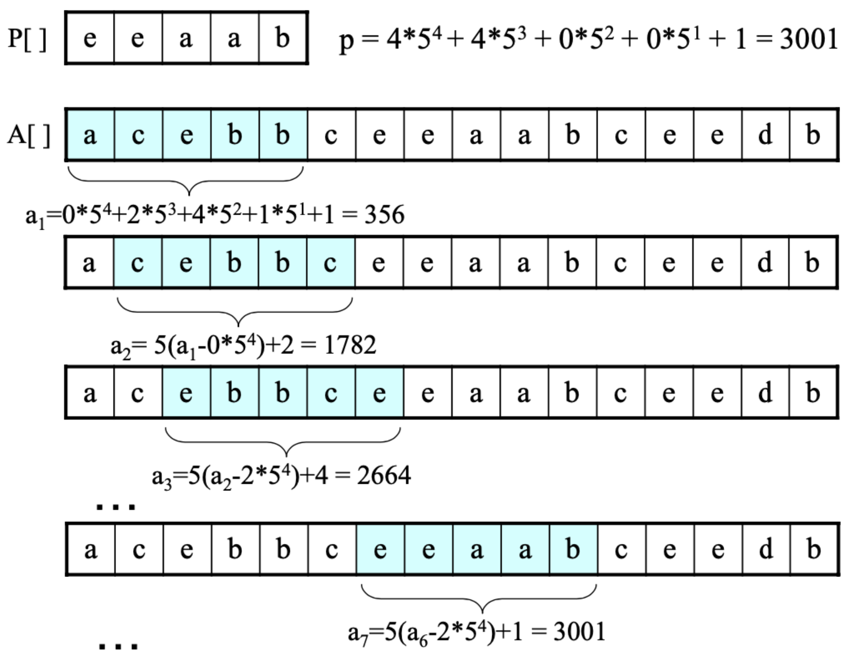
해결 : m의 크기에 상관없이 아래와 같이 계산할 수 있으므로 이 문제는 해결된다.
ai = d(ai-1 – dm-1A[i-1]) + A[i+m-1]

dm-1은 반복 사용되므로 미리 한번만 계산해 두면 된다.
곱셈 2회, 덧셈 2회로 충분하다.

예) “dcedabcc”
a1 = *354* + *253* + *452* + *351* + 0 ➔ “dceda”
a2 = 5*(a1 – 54*A[1])* + *A[6]* = *5(a1 – 54*3) + 1

즉, dceda를 나타내는 수치로부터 cedab를 나타내는 수치를 계산하는 데 상수 시간이 걸린다.

- 수치화 작업의 부담 알고리즘 예시




```
basicRabinKarp(A, P, d) {   // n : 배열 A의 길이, m : 배열 P의 길이
    p ← 0; a1 ← 0;
      for i←1 to m {        // 패턴P의 수치값 p와 a1 계산
        p ← dp + P[i];
        a1 ← da1 + A[i];
    }
    if (p = a1) then A[1] 자리에서 매칭이 일어났음을 알린다;
    for i ← 2 to n-m+1  {
      ai ← d(ai-1 – dm-1A[i-1]) + A[i+m-1];
      if (p = ai) then A[i] 자리에서 매칭이 일어났음을 알린다;
    }
}
```

=> 수행시간 : Θ(n)시간이 소요된다. 하지만 이것은 아직 라빈-카프 알고리즘이 아니다.

문제점 : 문자 집합 크기d와 패턴의 길이 m에 따라 ai가 매우 커질 수 있다. 심하면 컴퓨터 레지스터의 용량 초과된다. 오버플로우가 발생된다.

해결 : 나머지 연산(modulo)을 사용하여 ai의 크기를 제한한다.
ai = d(ai-1 – dm-1A[i-1]) + A[i+m-1] 대신
bi = (d(bi-1 – (dm-1 mod q) A[i-1]) + A[i+m-1]) mod q 사용한다.
q를 충분히 큰 소수(prime number)로 정한다. dq가 레지스터에 수용될 수 있는 크기로 한다.

나머지 연산을 이용한 매칭 예

```
RabinKarp(A,P,d,q) {    // n:배열A의길이, m:배열P의길이 {
    h ← 1;
    for i ← 1 to m-1
      h ← dh mod q;   // h 계산(= dm-1 mod q)
    p ← 0; b1 ← 0;
    for i ← 1 to m {
      p ← (dp + P[i]) mod q;    // 패턴 P의 수치값 p 계산
      b1 ← (db1 + A[i]) mod q;  // b1 계산
    }
    for i ← 1 to n-m+1{
      if (i ≠ 1) then bi ← (d(bi-1 – hA[i-1]) + A[i+m-1]) mod q;
          if (p = bi) then
            if (P[1...m] = A[i...i+m-1]) then // 진짜 매칭인지 알아봄
              A[i] 자리에서 매칭이 일어났음을 알린다;
}
```

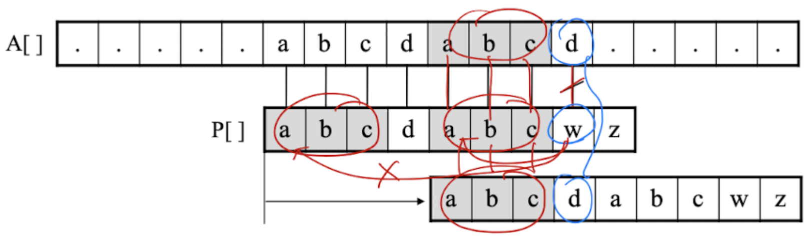
=> 수행시간 : 매칭 횟수가 상수 번이면 Θ(n) ➔ 평균 수행시간 Θ(n)시간이 소요된다.
=> 최악의 경우 Θ(mn)이 걸린다. 예를 들어 모든 문자가 동일한 경우이다.

KMP(Knuth-Morris-Pratt) 알고리즘

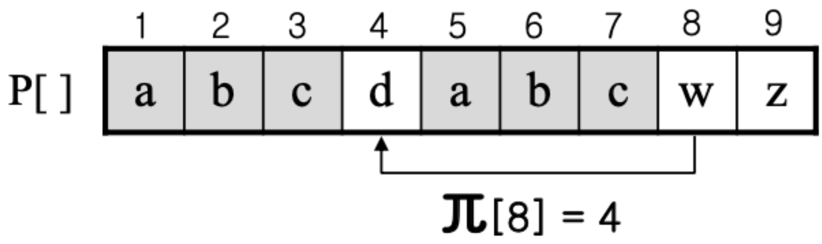
오토마타를 이용한 매칭과 비슷하다. 매칭에 실패했을 때 돌아갈 상태를 준비해둔다.

KMP 알고리즘의 전처리

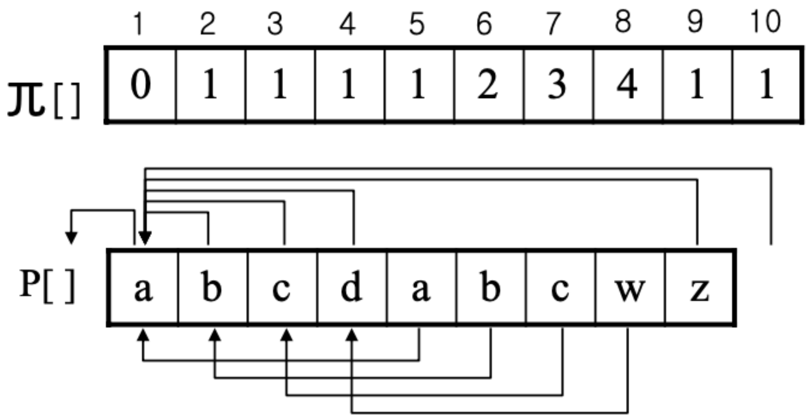
오토마타를 이용한 매칭보다 전처리 작업이 단순하다.



매칭이 실패했을 때 돌아갈 곳을 준비하는 작업이다.



back은 안하지만 그 자리에서 맴도는 현상은 있다. 하지만 수행시간에 영향이 있진 않다.



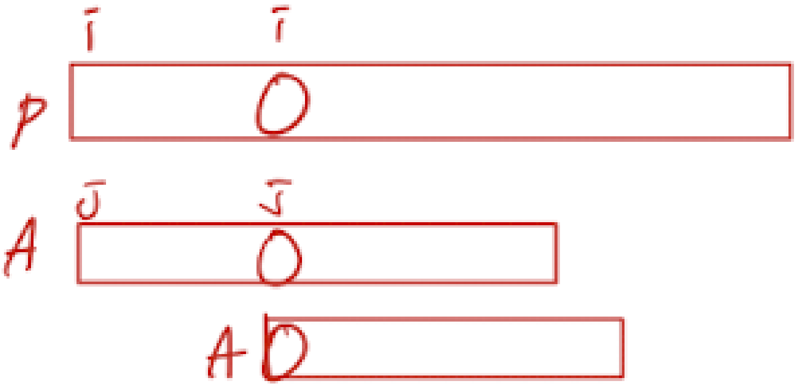
패턴의 각 위치에 대해 매칭에 실패했을 때 돌아갈 곳에 대한 정보 π를 준비해 둔다.

```
preprocessing(P[ ]) {
  j ←1;
  k ←0;
    π[1] ← 0;

    while (j ≤ m) {
      if (k = 0 or P[j] = P[k]) then { j++; k++; π[j] ← k; }
        else k ← π[k];
  }
}
```

=> 수행시간 : Θ(m)시간이 소요된다.

KMP 알고리즘

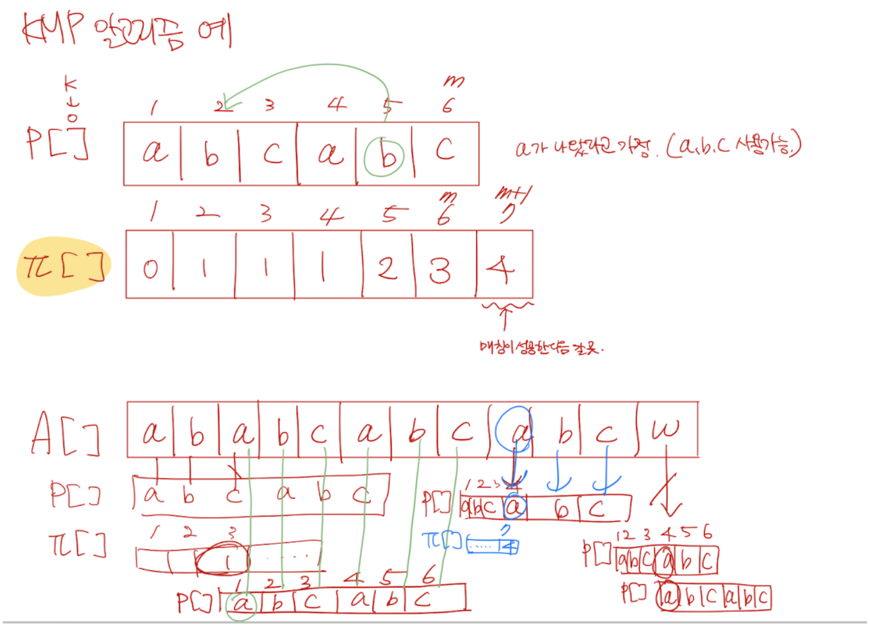



```
KMP(A[ ], P[ ]) { // n: 배열 A의 길이, m: 배열 P의 길이
    preprocessing(P);   // 수행시간 Θ(m)
    i ←1;          // 본문 문자열 포인터
    j ←1;          // 패턴 문자열 포인터

    while (i ≤ n) {
      if (j = 0 or A[i] = P[j]) then {
          i++; j++;
      } else j ← π[j];
      if (j = m+1) then {
        A[i-m]에서 매칭이 일어났음을 알린다;
        j ← π[j];
      }
    }
}
```

=> 수행시간 : Θ(n)이 소요된다.

KMP 알고리즘 예



보이어-무어(Boyer-Moore) 알고리즘

앞의 매칭 알고리즘들의 공통점

텍스트 문자열의 문자를 적어도 한번씩 훑는다. 따라서 최선의 경우에도 Ω(n)시간이 소요된다. (n보다 빠를 수 없다.)

보이어 무어 알고리즘의 특징

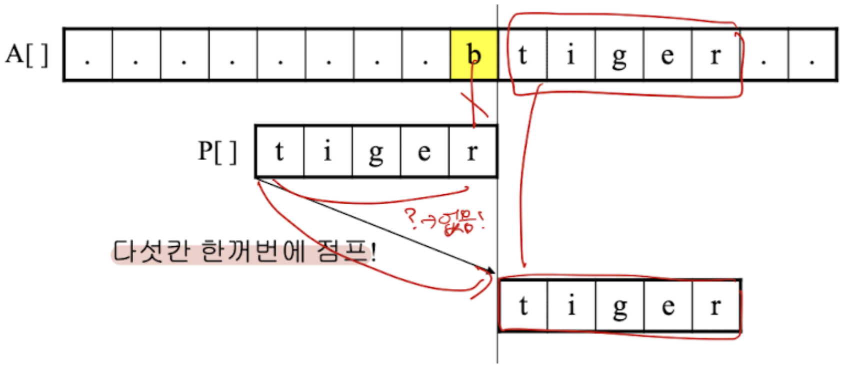
텍스트 문자를 다 보지 않아도 된다. 발상의 전환 -> 패턴의 오른쪽부터 비교를 한다. 매치될 가능성이 없으면 점프를 한다.

보이어 무어 호스풀 알고리즘 (Boyer-Moore-Horspool)

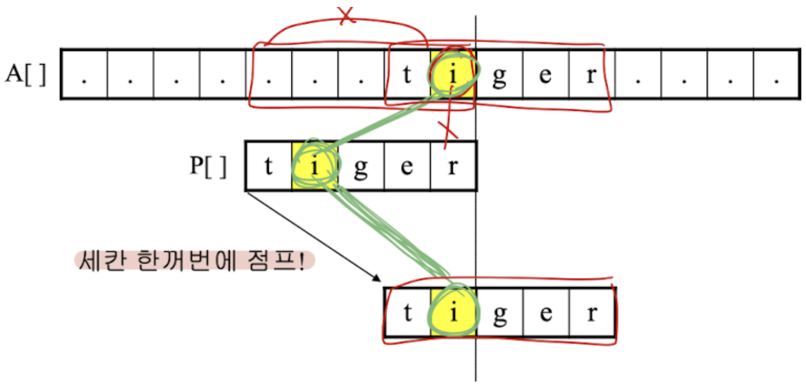
보이어-무어 알고리즘에서 까다로운 부분을 약식으로 처리한 알고리즘으로서, 약식으로 처리해도 전체 성능에는 거의 영향을 주지 않는다.
라빈-카프 알고리즘, KMP 알고리즘보다 평균적으로 빠르다.

보이어 무어 호스풀 알고리즘 관찰

상황 : 텍스트의 b와 패턴의 r을 비교하여 실패했다.



상황 : 텍스트의 i와 패턴의 r을 비교하여 실패했다.



보이어 무어 호스풀 알고리즘 점프 정보 준비 (preprocessing)

5

점프정보를 찾는시간

- 패턴 "tiger"에 대한 점프 정보

오른쪽 끝문자	t	i	g	e	r	기타
jump	4	3	2	1	5	5

8

- 패턴 "rational"에 대한 점프 정보

오른쪽 끝문자	r	a	t	i	o	n	a	l	기타
jump	7	~~6~~	5	4	3	2	1	8	8

오른쪽 끝문자	r	t	i	o	n	a	l	기타
jump	7	5	4	3	2	1	8	8

보이어 무어 호스풀 알고리즘

```
BoyerMooreHorspool(A[ ], P[ ]) {     //  n : 배열 A의 길이, m : 배열 P의 길이
    computeJump(P, jump);          // preprocessing : 수행시간 Θ(m)
    i ← 1;
    while (i ≤ n - m+1) {
```

```
        j←m; k←i+m-1;
        while ( j > 0 and P[j] = A[k]) {
          j--; k--;
         }
        if (j = 0) then A[i] 자리에서 매칭이 일어났음을 알린다;
        i ← i + jump[A[i + m - 1]];   // jump 정보 얻는 시간 θ(1)
      }
}
```

=> 수행시간 : 최악의 경우 수행시간은 Θ(mn)시간이 소요된다. 최선의 경우 수행시간은 Θ(n/m)시간이 소요된다. 즉, 입력에 따라 다르지만 일반적으로 Θ(n)보다 시간이 덜 든다.

- 최악의 경우 예

모든 문자가 동일한 경우
A=aaaaa..............aaa =a^n
P = aaaa......aa = a^m

- 최선의 경우 예

A = abcdybbbbkcccctddddx
P = abcde

요약

1. 원시적인 매칭 알고리즘은 텍스트의 각 위치에서 시작해 패턴 문자열과 일치하는지 체크하는 방법이다. 수행시간은 O(mn)이다.
2. 오토마타를 이용하는 알고리즘은 매칭 과정에서 불일치가 일어났을 때 처음부터 다시 비교하지 않고 문맥상의 정보를 이용해 중간부터 비교할 수 있도록 한다. 수행시간은 Θ(n)이다.
3. 라빈-카프 알고리즘은 패턴을 수치화해 (문자열 비교 -> 수치 비교로 전환) 문자열 매칭을 수행한다. 평균 수행시간은 Θ(n)이다. 최선 수행시간은 Θ(n)이며 최악 수행시간은 Θ(mn)이다.
4. KMP 알고리즘은 매칭 과정에서 불일치가 일어났을 때 처음부터 다시 비교하지 않고 중간부터 비교할 수 있도록 되돌아 갈 위치를 비열로 나타낸다. 수행시간은 Θ(n)이다.
5. 보이어-무어-호스풀 알고리즘은 패턴의 뒷부분에 대응되는 텍스트의 문자를 이용해 텍스트를 보지 않고도 뛰어넘을 수 있게한다. 최악의 경우 시간은 O(mn)이지만 평균적으로 가장 좋은 성능을 보인다. 최선 평균시간 Θ(n/m)이고 평균 수행시간은 Θ(n)보다는 Θ(n/m)에 가깝다.

문자열 매칭 알고리즘

텍스트 길이 n
패턴 길이 m

알고리즘	매칭작업 전에 실행 preprocessing time	matching time	특징
Naive(brute force)	0	$O(mn)$	매우 원시적
Finite automaton	P $O(m\|\Sigma\|)$	$\Theta(n)$	오토마타 이용 Σ : 입력 알파벳
Rabin-Karp	P b, $\Theta(m)$	최악 $\Theta(mn)$ √최선 $\Theta(n)$ √평균 $\Theta(n)$	패턴을 수치화
Knuth-Morris-Pratt	π $\Theta(m)$	$\Theta(n)$	부분패턴에 깃든 효용성을 최대한 이용
Boyer-Moore-Horspool	jump $\Theta(m)$	최악 $\Theta(mn)$ 최선 $\Theta(n/m)$ 평균 $\Theta(n)$보다는 $\Theta(n/m)$에 가까움	텍스트 문자열을 보지 않고 점프할 수 있는 기회를 최대화