4장 스트링 처리 알고리즘

목차

- 스트링 탐색 알고리즘
  - 직선적 알고리즘
  - KMP 알고리즘
  - 보이어-무어 알고리즘
  - 라빈-카프 알고리즘
- 패턴 매칭 알고리즘
- 화일 압축 알고리즘
  - 런-길이 인코딩
  - 가변-길이 인코딩
- 암호화 알고리즘
  - 단순한 기법
  - 공개 키 암호화 시스템

스트링 탐색 알고리즘

- 문서 작성 시
  - 텍스트(text) : 문서
  - 패턴(pattern) : 탐색할 스트링
  - 스트링(string)
    - 문자가 연속적으로 나열된 것
    - 텍스트(text) 스트링
    - 이진(binary) 스트링
- 스트링 탐색 알고리즘의 설계
  - 필연적으로 잘못된 시작(false start) 발생
    - 불일치가 발생한 위치까지 비교한 0개 이상의 문자를 의미
  - 잘못된 시작의 횟수와 길이를 줄이는 것

직선적 알고리즘(1)

- 한 글자 또는 한 비트씩 오른쪽으로 진행
- 텍스트의 처음부터 끝까지 모두 비교하며 탐색하는 알고리즘

```
bruteForce(p[], t[])
  M ← 패턴의 길이; N ← 텍스트의 길이;
  for (i ← 0, j ← 0; j  < M and i < N; i ← i + 1, j ← j + 1) do
    if (t[i] ≠ p[j]) then {
      i ← i − j;
      j ← −1;
    }
  if (j = M) then return i − M;
  else return i;
end bruteForce()
```

직선적 알고리즘(2)

- 시간 복잡도
  - 최악의 경우 시간 복잡도는 텍스트의 모든 위치에서 패턴을 비교해야 하므로 $O(MN)$이 됨

직선적 스트링 탐색 과정

1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	0
1	0	1	0	0	1	1	1																	
	1	0	1	0	0	1	1	1																
		1	0	1	0	0	1	1	1															
			1	0	1	0	0	1	1	1														
				1	0	1	0	0	1	1	1													
					1	0	1	0	0	1	1	1												
						1	0	1	0	0	1	1	1											
							1	0	1	0	0	1	1	1										
								1	0	1	0	0	1	1	1									
									1	0	1	0	0	1	1	1								
										1	0	1	0	0	1	1	1							
											1	0	1	0	0	1	1	1						
												1	0	1	0	0	1	1	1					
													1	0	1	0	0	1	1	1				
														1	0	1	0	0	1	1	1			
															1	0	1	0	0	1	1	1		
																1	0	1	0	0	1	1	1	
1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	0

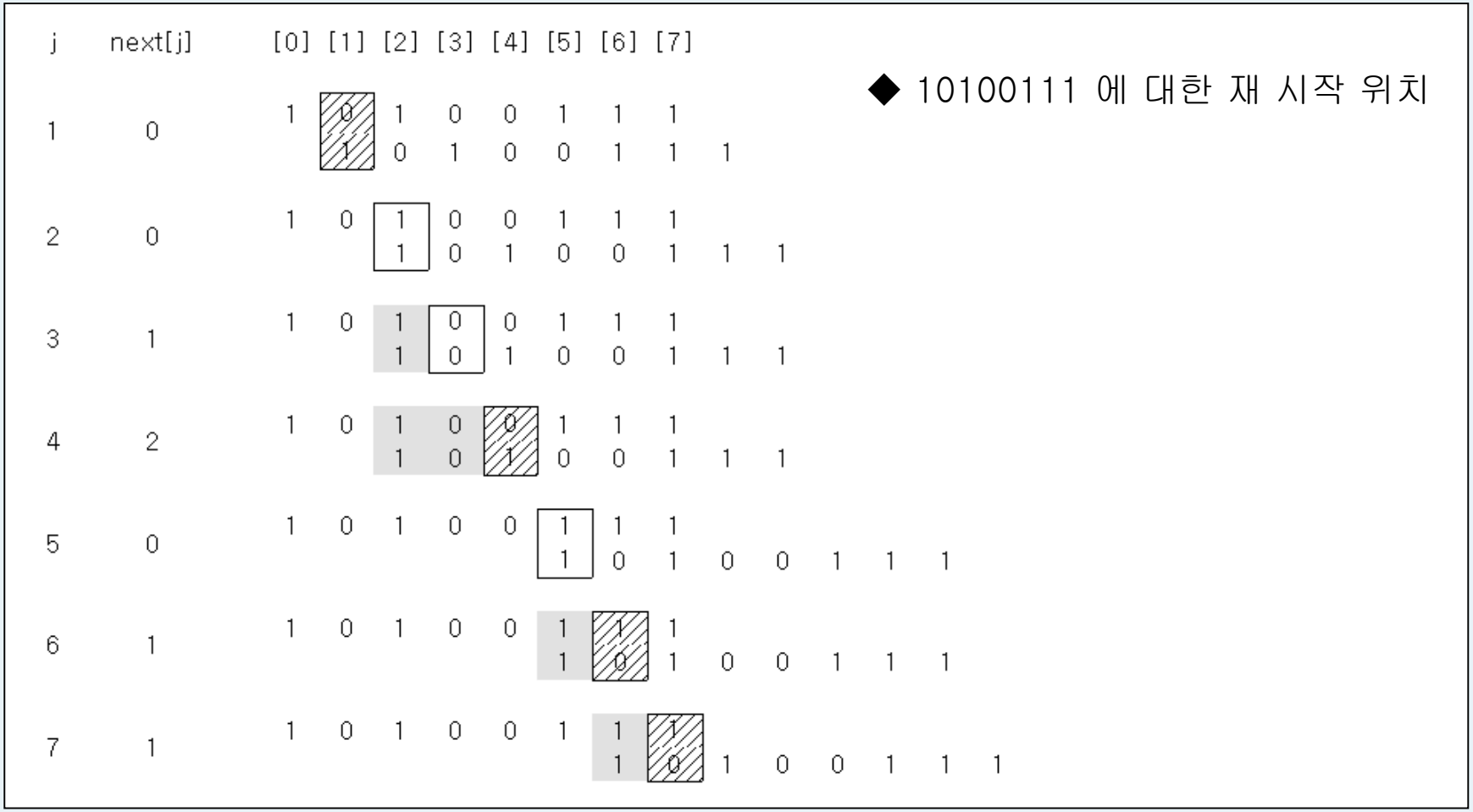
KMP 알고리즘(1)

- KMP : Knuth, Morris and Pratt
- 불일치가 발생한 텍스트 스트링의 앞 부분에 어떤 문자가 있는지를 미리 알고 있으므로, 불일치가 발생한 앞 부분에 대하여 다시 비교하지 않고 매칭을 수행
- 패턴을 전처리하여 배열 next[M]을 구해서 잘못된 시작을 최소화함
  - next[M] : 불일치가 발생했을 경우 이동할 다음 위치
- 시간 복잡도 : $O(M+N)$

KMP 알고리즘(2)

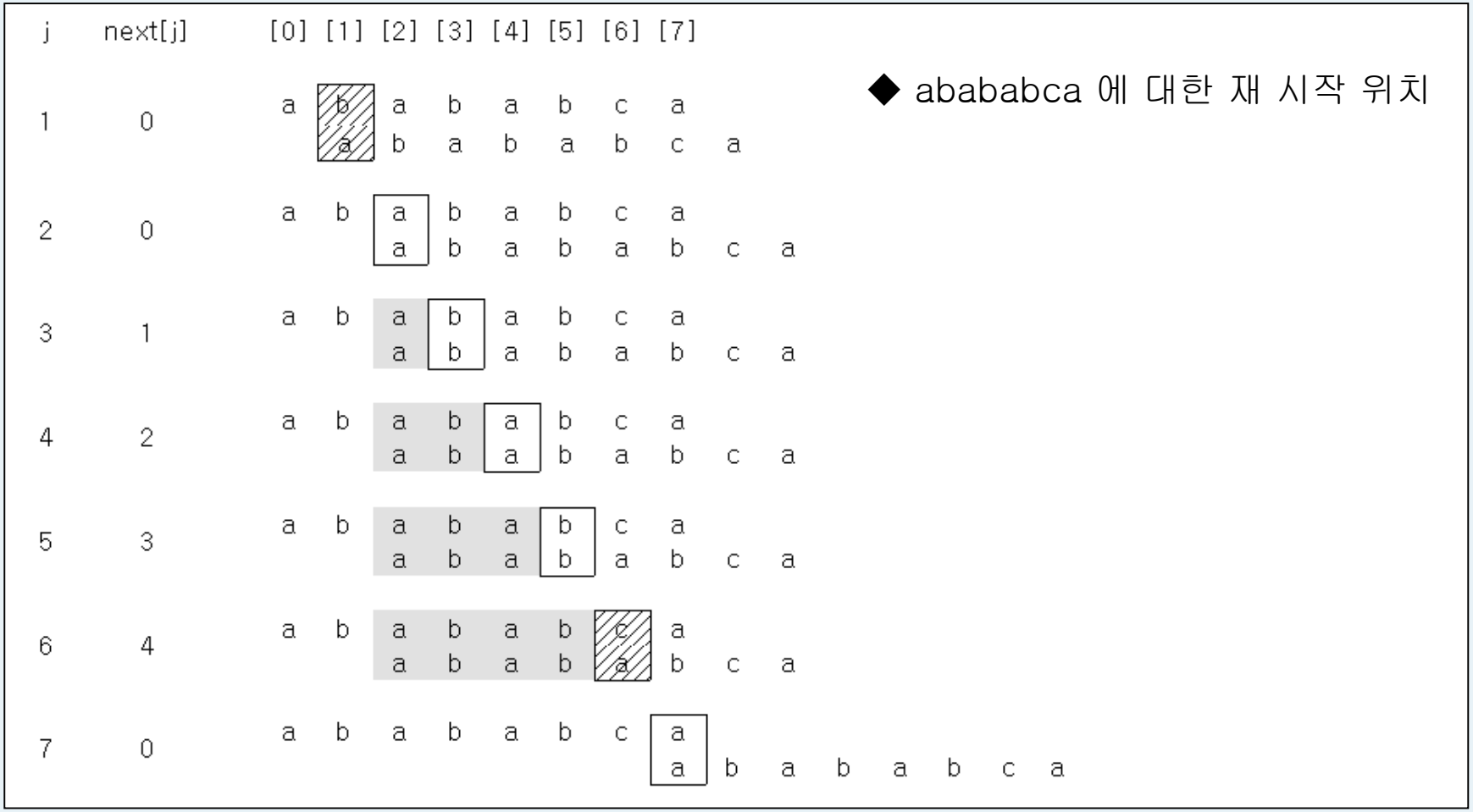
```
KMP(p[], t[])
  M ← 패턴의 길이; N ← 텍스트의 길이;
  iInitNext(p);
  for (i ← 0, j ← 0; i  < M and i < N; i ← i + 1, j ← j + 1) do
    while ((j ≥ 0) and (t[i] ≠ p[j])) do
      j ← next[j];
  if (j = M) then return i - M;
  else return i;
end KMP()
```

재 시작 위치(1)



◆ 10100111 에 대한 재 시작 위치

재 시작 위치(2)



◆ abababca 에 대한 재 시작 위치

재 시작 위치 알고리즘

```
initNext(p[])
  M ← 패턴의 길이;
  next[0] ← -1;
  for (i ← 0, j ← -1; j  < M; i ← i + 1, j ← j + 1) do {
    next[i] ← j;
    while ((j ≥ 0) and (p[i] ≠ p[j])) do
      j ← next[j];
  }
end initNext()
```

패턴이 내장된 KMP 알고리즘(1)

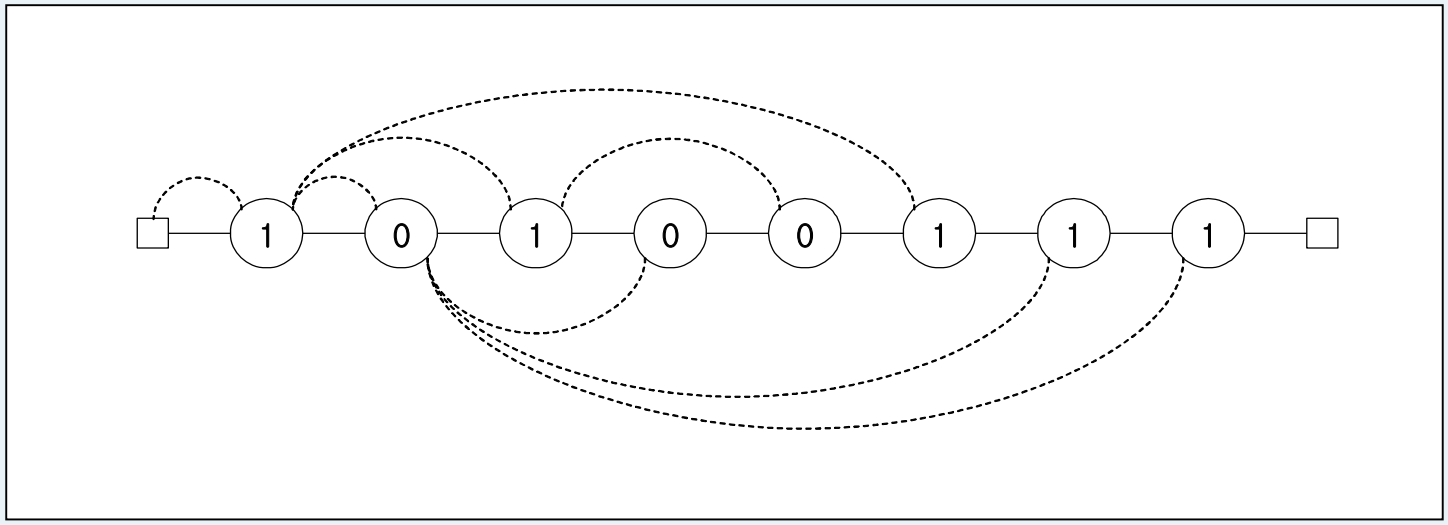
```
KMP(t[])
    i ← -1;
    sm : i ← i + 1;
    s0 : if (t[i] ≠ '1') then goto sm; i ← i + 1;
    s1 : if (t[i] ≠ '0') then goto s0; i ← i + 1;
    s2 : if (t[i] ≠ '1') then goto s0; i ← i + 1;
    s3 : if (t[i] ≠ '0') then goto s1; i ← i + 1;
    s4 : if (t[i] ≠ '0') then goto s2; i ← i + 1;
    s5 : if (t[i] ≠ '1') then goto s0; i ← i + 1;
    s6 : if (t[i] ≠ '1') then goto s1; i ← i + 1;
    s7 : if (t[i] ≠ '1') then goto s1; i ← i + 1;
    return i-8;
end KMP()
```

패턴이 내장된 KMP 알고리즘(2)

- 유한 상태 장치(finite state machine: FSM)
  - 상태(state; 원으로 표시)
  - 전이(transition; 선으로 표시)
    - 일치 전이(match transition; 실선으로 표시) –오른쪽으로 이동
    - 불일치 전이(non-match transition; 점선으로 표시) – 왼쪽으로 이동
  - 시작점(왼쪽 끝의 사각형)
  - 종료점(오른쪽 끝의 사각형)

패턴이 내장된 KMP 알고리즘(3)

- KMP 알고리즘을 위한 유한 상태 장치

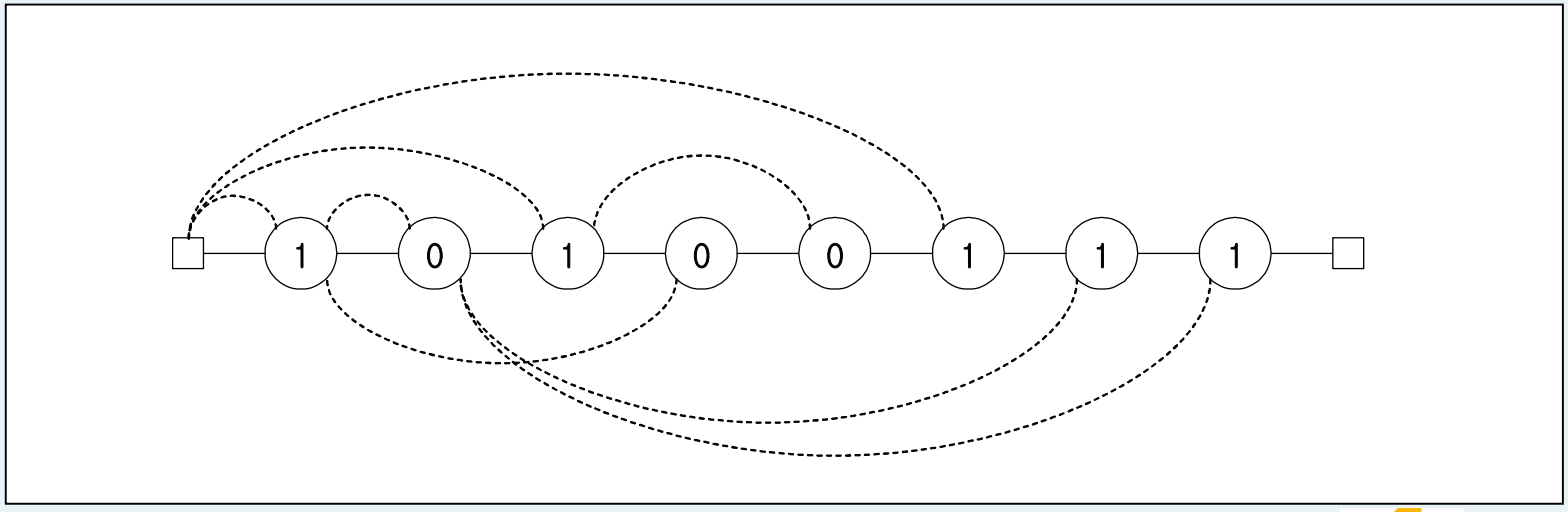


개선된 유한 상태 장치(1)

- initNext 알고리즘의 next[i] ← j; 변경

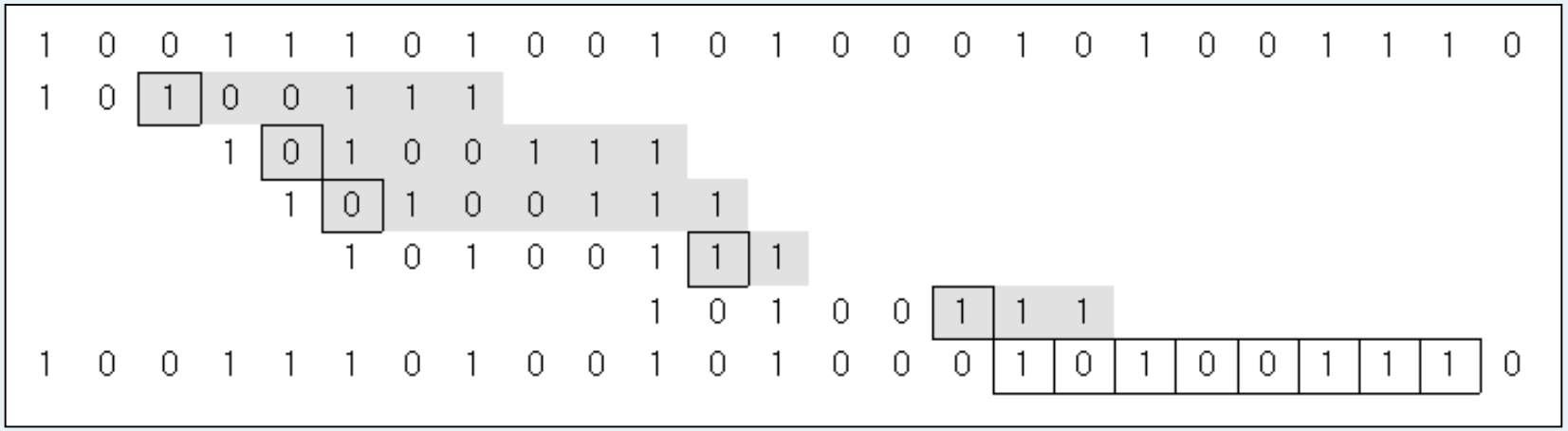
```
if (p[i] = p[j]) then next[i] ← next[j];
else next[i] ← j;
```

- 개선된 유한 상태 장치



개선된 유한 상태 장치(2)

- 개선된 유한 상태 장치를 이용한 KMP 알고리즘 수행 과정
  - 10100111

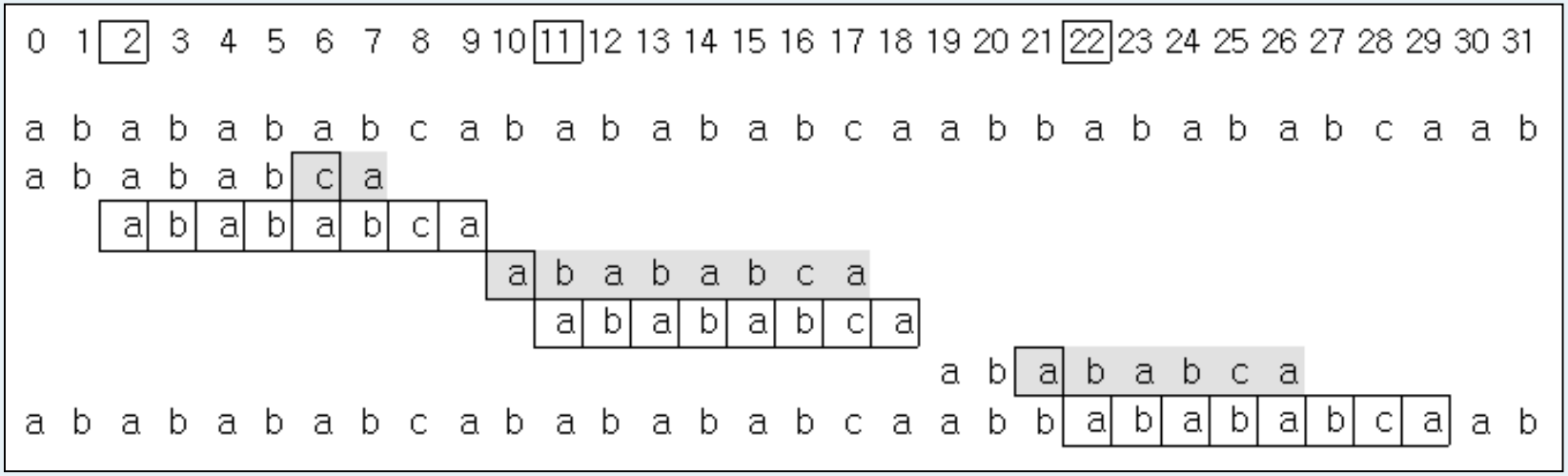


- *M*+*N* 번 이상의 비교를 하지 않으므로 시간 복잡도는 O(*M*+*N*)

개선된 유한 상태 장치(3)

- 개선된 유한 상태 장치를 이용한 KMP 알고리즘 수행 과정
  - abababca

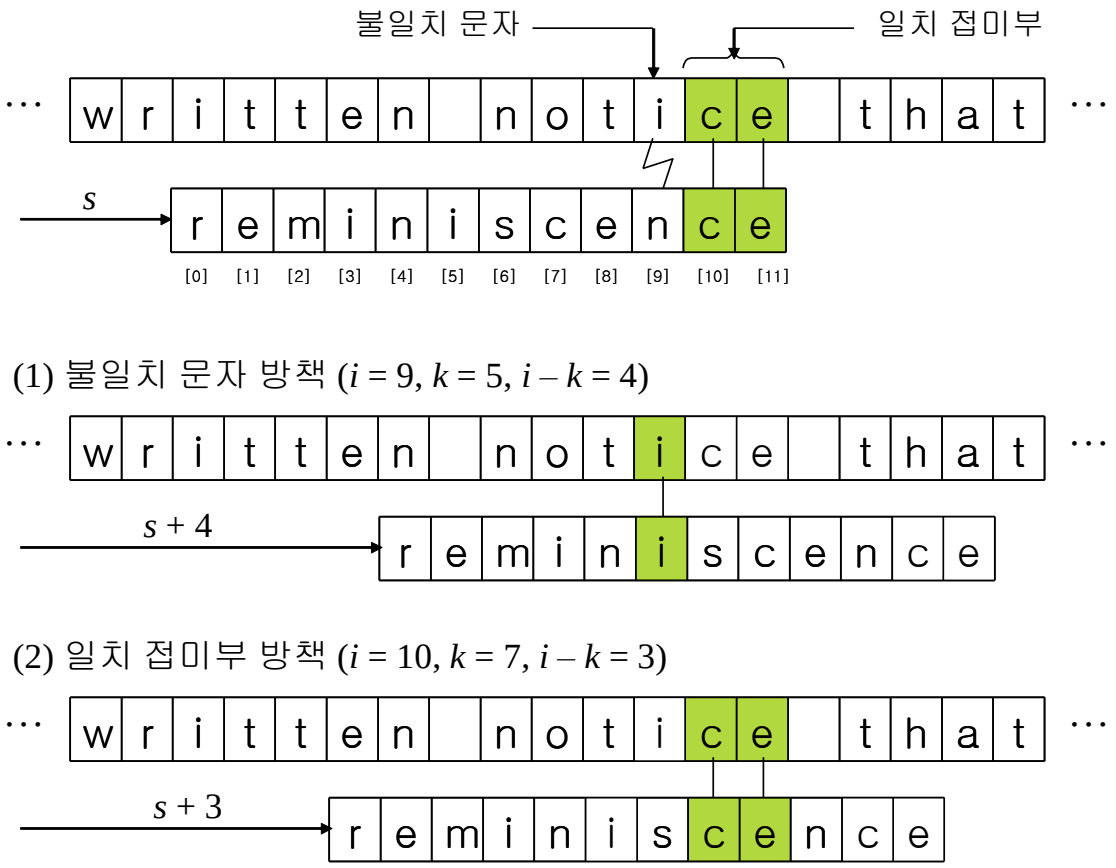
j	1	2	3	4	5	6	7
next[j]	0	-1	0	-1	0	4	-1



보이어-무어 알고리즘

- 오른쪽에서 왼쪽으로 스트링 탐색을 진행
- 불일치 문자 방책(mismatched character heuristic) 사용
  - 텍스트에 있는 불일치가 발생한 문자가 패턴의 문자가 일치하도록 패턴을 오른쪽으로 이동
- 일치 접미부 방책(good suffix heuristic) 사용
  - 패턴에서 불일치가 발생한 문자의 오른쪽에 있는 최대 접미부가 일치하도록 패턴을 오른쪽을 이동하는 것
- 두 방법 중 패턴을 우측으로 이동하는 거리가 더 긴 것을 선택
- *M+N* 번 이상의 문자 비교를 하지 않으며, 알파벳이 작지 않고, 패턴이 길지 않을 때는 약 *N/M* 단계 만에 수행

불일치 문자 방책과 일치 접미부 방책



불일치 문자 방책 알고리즘(1)

```
BM(p[], t[])
  M ← 패턴의 길이; N ← 텍스트의 길이;
  initSkip(p);
  for (i ← M-1, j ← M-1; j  ≥ 0; i ← i - 1, j ← j - 1) do
    while (t[i] ≠ p[j]) do {
      k ← skip[index(t[i])];
      if (M-j > k) then i ← i + M - j;
      else i ← i + k;
      if (i ≥ N ) then return N;
      j ← M - 1;
    }
  return i+1;
end BM()
```

불일치 문자 방책 알고리즘(2)

- initSkip() 함수 : skip 배열을 만듦

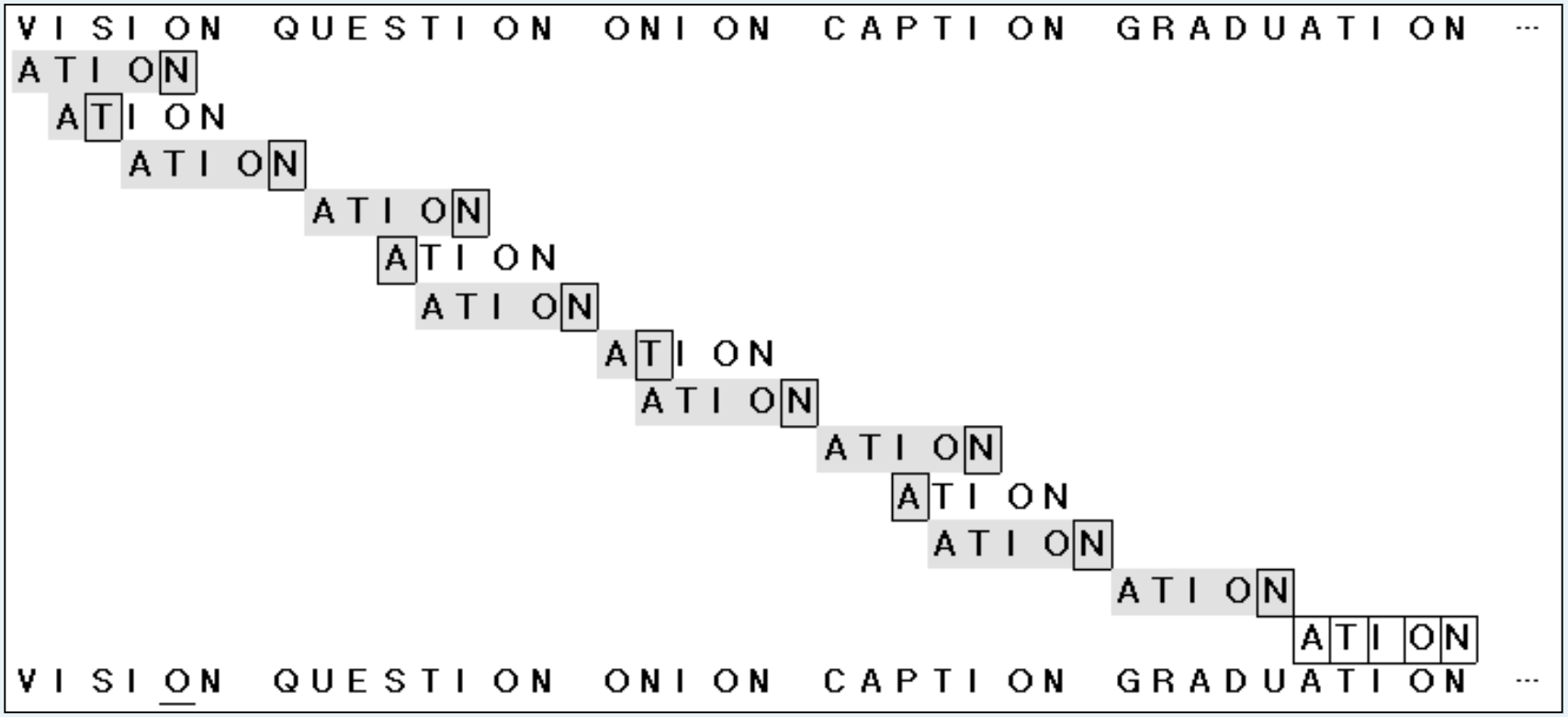
```
initSkip(p[])
 M ← 패턴의 길이;
 for (i ← 0; i < NUM; i ← i+1) do skip[i] ← M;
 for (i ←  0; i < M; i ← i+1) do skip[index(p[i])] ← M - i - 1;
end initSkip()
```

- ATION 일 때의 skip 배열

N	O	I	T	A	다른 모든 문자
0	1	2	3	4	5

불일치 문자 방책 알고리즘(3)

- 불일치 문자 방책을 사용한 보이어-무어 스트링 탐색 과정



라빈-카프 알고리즘(1)

- 스트링을 숫자값으로 바꾼 다음 해시 값을 계산하여 매칭하는 알고리즘
- 최악의 시간 복잡도는 $O(MN)$이지만 평균적으로는 선형에 가까운 $O(M+N)$의 시간 복잡도를 보여줌
- 이론적인 배경
  - 첫 번째 M 자리의 나머지를 구한 다음부터는 한 자리씩만 추가하면서 간단한 계산으로 나머지를 구할 수 있으므로 빠른 시간에 스트링 탐색을 수행

호너의 방법

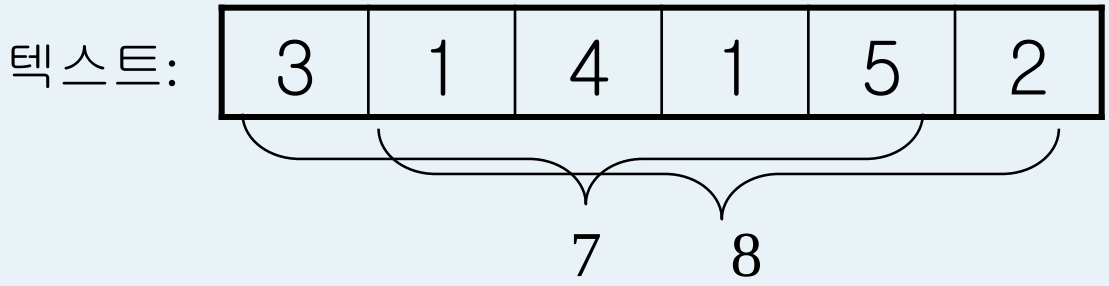
- 패턴 P[M]에 대한 10진수 p의 계산 과정

$$p = P[0]\cdot 10^{m-1} + P[1]\cdot 10^{m-2} + \cdots + P[m-1]\cdot 10^{0}$$
$$= P[m-1] + 10(P[m-2] + 10(P[m-3] + \cdots + 10(P[1] + 10P[0])\cdots))$$

- 패턴 P[M] = 31415라고 할 때, p를 구하는 과정

$$p = 31415$$
$$= 3\cdot 10^{4} + 1\cdot 10^{3} + 4\cdot 10^{2} + 1\cdot 10^{1} + 5\cdot 10^{0}$$
$$= 5 + 10(1 + 10(4 + 10(1 + 10\cdot 3)))$$

계산 예



d = 10, q = 13, m = 5

D = 10^4 mod 13 = 3

14152 mod 13 = ((31415 - 3·10^4)10 + 2) mod 13

= ((7 + 10·13 - 3·3)10 + 2) mod 13

= (11·10 + 2) mod 13

= 8

- 31415 mod 13 = 7
- 128 mod 13 = 11
- 112 mod 13 = 8

라빈-카프 알고리즘(2)

```
RK(p[], t[])
  dM ← 1; h1 ← 0; h2 ← 0;
  M ← 패턴의 길이; N ← 텍스트의 길이;
  for (i ← 1; i  < M; i ← i + 1) do
      dM ← (d*dM) mod q;
  for (i ← 0; i  < M; i ← i + 1) do {
      h1 ← (h1 * d + index(p[i])) mod q;
      h2 ← (h2 * d + index(t[i])) mod q;
  }
  for (i ← 0; h1 ≠ h2; i ← i + 1) do {
      h2 ← (h2 + d * q − index(t[i]) * dM) mod q;
      h2 ← (h2 + d + index(t[i+M])) mod q;
      if (i > N−M) then return N;
  }
  return i;
end RK()
```

패턴 매칭 알고리즘

- 패턴 매칭(pattern matching)
  - 텍스트 스트링에서 원하는 문자 패턴을 찾아내는 것
  - 패턴 기술
    - ① 접합(concatenation)
      - 패턴에서 인접해 있는 두 문자가 텍스트에서 나타나면 매치
    - ② 논리합(or)
      - 두 개의 문자 중 하나가 텍스트에 나타나면 매치
    - ③ 폐포(closure)
      - 특정한 문자가 0개 이상 나타나면 매치

정규식(regular expression)

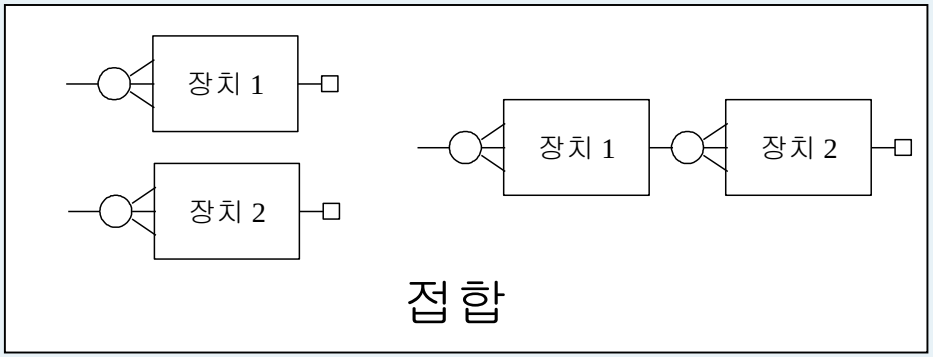
- 세 가지 기본 연산들로 이루어진 심볼들의 스트링
- 심볼(symbol)
  - * : 괄호 안에 있는 문자들이 0번 이상 나타남
  - ? :어떤 문자와도 매치됨
- 예
  - ?*(ie+ei)?* : *ie* 또는 *ei* 를 가지고 있는 모든 단어
  - (1+01)*(0+1) : 연속적으로 두 개의 0이 나오지 않는 0과 1로 이루어진 모든 스트링

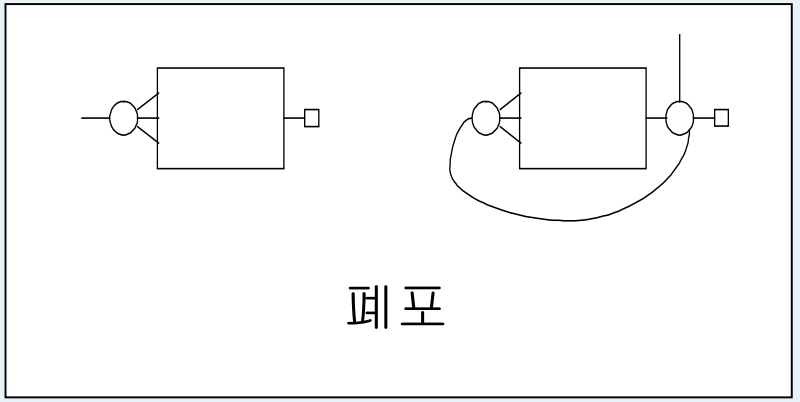
패턴 매칭 장치

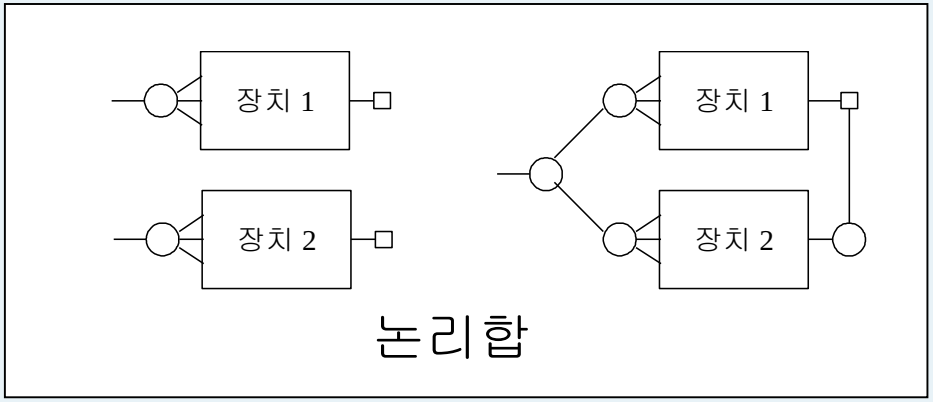
- 패턴 매칭 장치(pattern matching machine)
  - 패턴 매칭에 사용되는 장치 패턴
  - 결정적(deterministic) 장치
    - 각각의 상태 전이가 다음 입력 문자에 의해 완전하게 결정되는 것
    - 예 : KMP 알고리즘을 위한 유한 상태 장치
  - 비결정적(nondeterministic) 장치
    - 패턴을 매치하기 위해 하나 이상의 방법이 있을 경우 장치가 올바른 것을 찾아 나가는 것
    - 텍스트 스트링에서 (A*B+AC)D와 같은 정규식을 찾는 경우 사용되며, 유일한 시작 상태와 종료 상태를 가진다.

패턴 매칭 장치 구성 방법

◆ 패턴 매칭 구성 장치

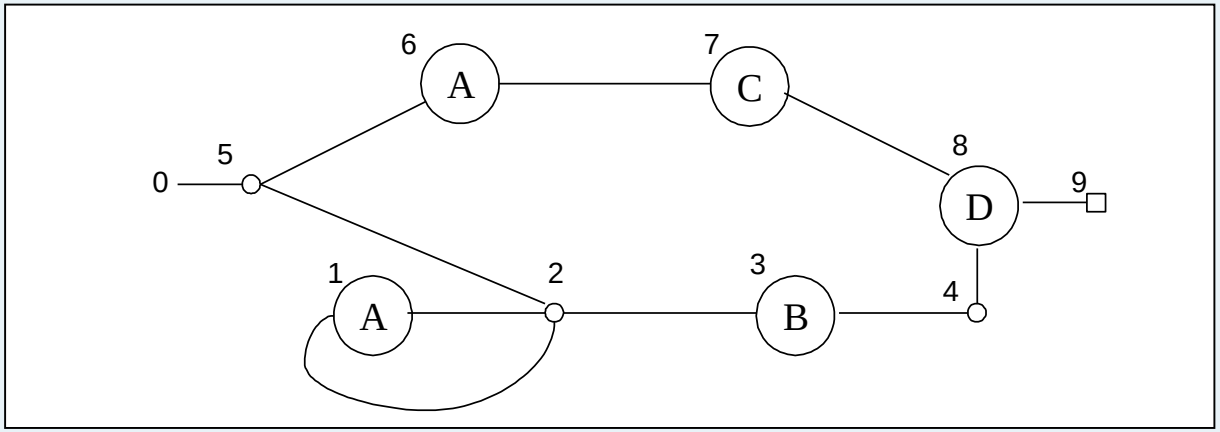






패턴 매칭 장치의 예

- (A*B+AC)D를 위한 비결정적 패턴 매칭 장치



- 패턴 매칭 장치의 배열 표현

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
ch		A		B			A	C	D	
next1	5	2	3	4	8	6	7	8	9	0
next2	5	2	1	4	8	2	7	8	9	0

패턴 매칭 알고리즘 구현

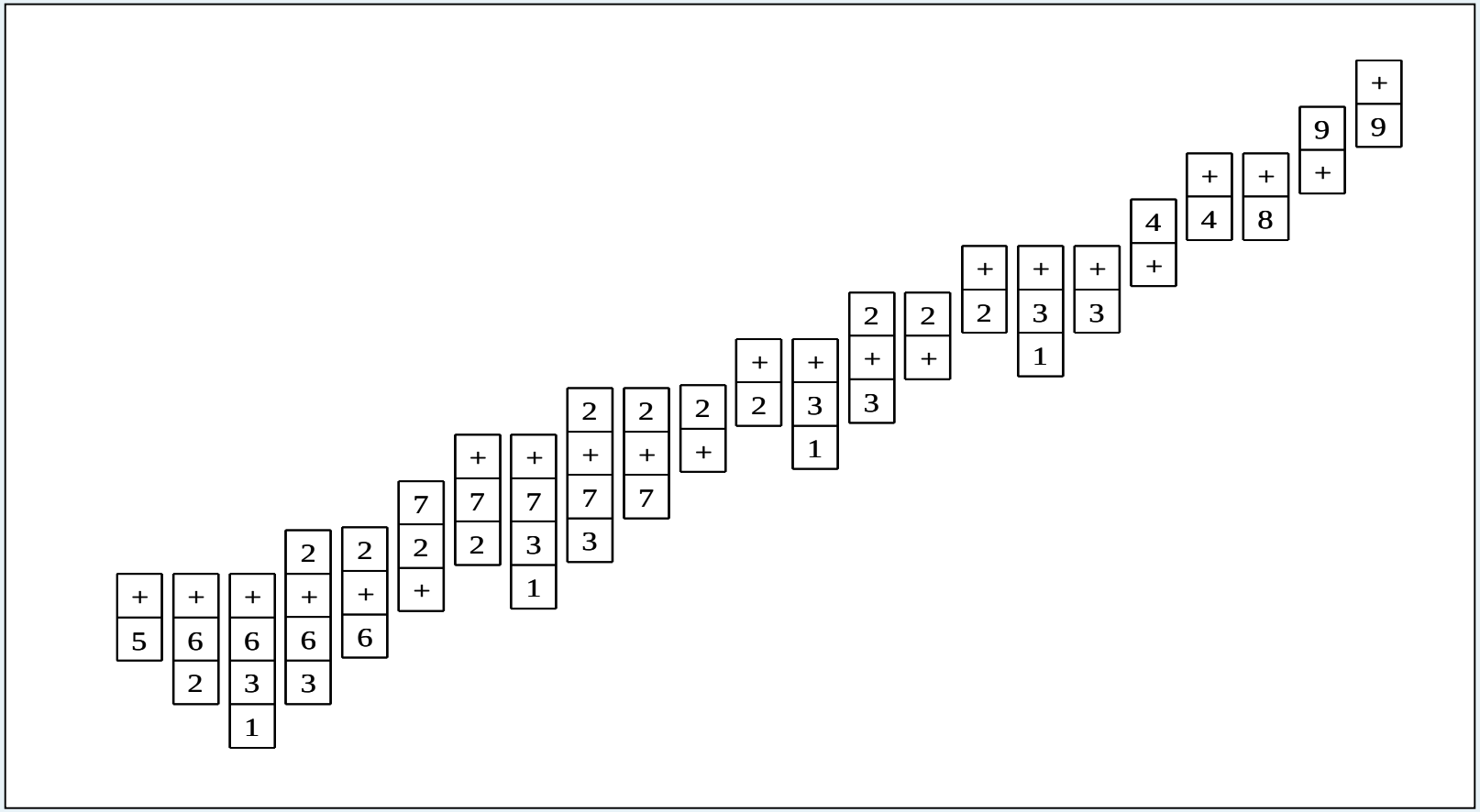
- 패턴 매칭 장치를 구현
  - 모든 가능한 매치를 체계적으로 조사
  - 장치를 구현하는데 가장 적합한 자료구조
  - 데크(deque: double-ended queue)를 사용
    - 스택과 큐의 특징을 조합
    - 양방향에서 항목을 추가하는 것이 가능
    - 입력은 양방향에서 가능
    - 삭제는 데크의 처음에서만 가능한 출력-제한 데크 (output-restricted deque)로 사용
- N 문자로 이루어진 텍스트 스트링에서 *M*-상태 장치가 패턴을 찾는 것은 최악의 경우에도 *NM* 번 이하의 상태 전이만이 필요

동작 과정

- 문자가 매치됨 → 새로운 상태를 데크의 끝에 삽입 (insertLast)
- 상태가 비어 있음 → 두 개의 가능한 상태를 데크의 처음에 삽입(insertFirst)
- scan을 만남 → 입력 스트링에 대한 포인터를 다음 문자로 이동
- 종료 조건
  - 입력의 끝까지 갔을 때 (매치되지 않음)
  - 상태 0을 만남 (매치됨)
  - 데크에 scan 마크 하나만 남음 (매치되지 않음)

패턴 매칭 알고리즘 수행 예

- AAABD를 식별하는 과정



화일 압축 알고리즘

- 화일 압축(file compression)
  - 시간보다는 공간을 절약하는 방법에 대해 연구
  - 대부분의 컴퓨터 화일에서 데이터가 중복되어 있다는 것에 착안
  - 대상
    - 텍스트 파일
    - 인코딩된 이미지의 래스터 파일
    - 사운드나 다른 아날로그 신호의 디지털 표현 등

특성

- 일반적인 압축 방법은 압축되지 않고 원래보다 길이를 더 길게 만드는 화일이 존재함
- 화일 압축 기법은 중요하지 않음
  - 컴퓨터 저장장치의 가격이 극적으로 하락하고 있음
- 화일 압축 기법은 더 중요해 짐
  - 파일의 크기가 점점 더 커지고 있으므로 약간의 압축으로도 많은 양을 절약할 수 있음

런-길이 인코딩 (1)

- 런-길이 인코딩(run-length encoding)
  - 동일한 문자가 여러 개 나올 경우 숫자와 문자 쌍으로 화일을 압축하는 기법
  - 이진수로 표현되는 비트맵 이미지를 압축하는 데 사용

런-길이 인코딩 (2)

- 화일에서 가장 기본적인 중복은 같은 문자가 연속적으로 나타나는 것임
  - AAAABBCCC → 4A2B3C
  - 000001111000 → 6 4 3
- 문자를 사용한 화일 압축 기법은 숫자와 문자가 혼합되어 있는 화일에는 적용할 수 없음
  - 텍스트에서 드물게 나타나는 문자를 탈출 문자(escape character) 로 사용
  - 탈출 순차(escape sequence) : 탈출 문자, 개수, 1개의 반복 문자
  - AAAABBCCCCC → QDABBQEC
  - Q : Q<space>
  - 50개의 B : QZBQXB

가변-길이 인코딩

- **가변-길이 인코딩**(variable-length encoding)
  - 자주 나타나는 문자에는 적은 비트를 사용하고 드물게 나타나는 문자에는 많은 비트를 사용하여 인코드(encode)하는 압축 기법
- ABRACADABRA 에 다섯 비트를 사용

문자	A	B	C	D	R
빈도수	5	2	1	1	2
코드	00001	00010	00011	00100	10010

  - 0000100010100100000100011000010010000001000101001000001
- ABRACADABRA에 대한 코드 할당

문자	A	B	C	D	R
빈도수	5	2	1	1	2
코드	0	1	10	11	01

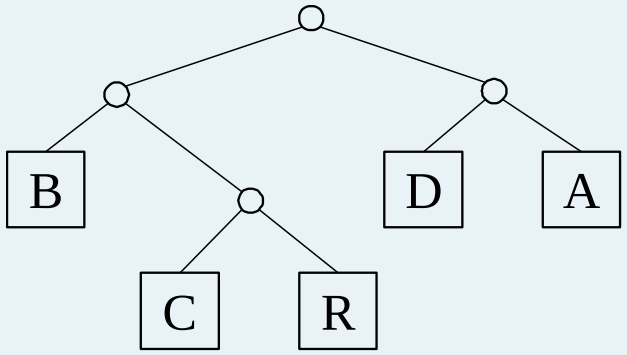
  - 0 1 01 0 10 0 11 0 1 01 0

트라이(1)

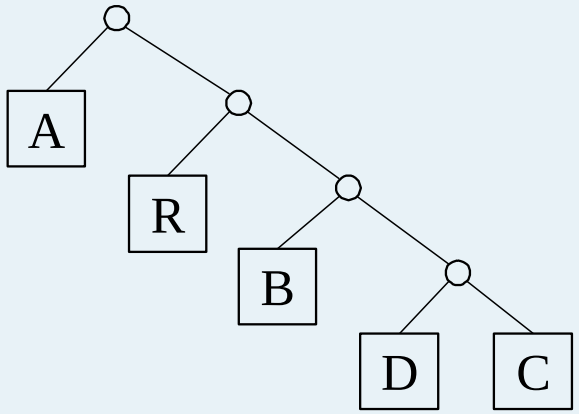
- 트라이(trie)
  - 하나의 문자 코드가 다른 문자 코드의 접두부가 되지 않는 것을 보장하는 자료구조
  - 비트 스트링을 유일하게 디코드 할 수 있음
  - 하나의 문자열로부터 여러 개의 트라이를 생성할 수 있음
    - 여러 개의 트라이 중에서 어떤 것이 가장 효율적인 것인지를 결정하는 방법이 있어야 함

트라이(2)

- ABRACADABRA에 대한 두 가지 인코딩 트라이
  - A: 11, B: 00, C: 010
    D: 10, R: 011



  - A: 0, R: 10, B: 110
    D: 1110, C: 1111



허프만 인코딩

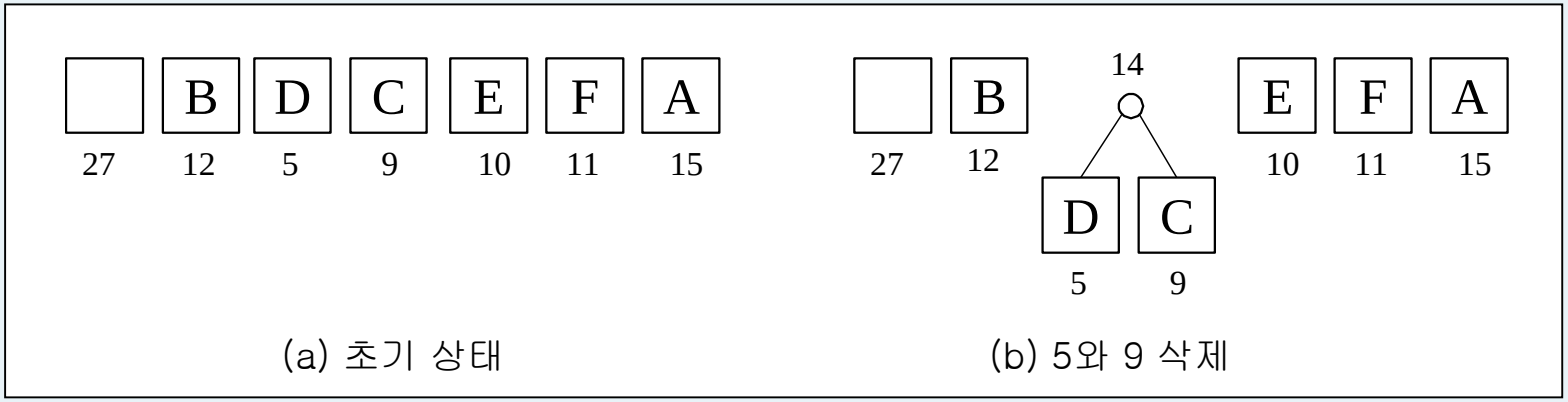
- 허프만 인코딩(huffman encoding)
  - 여러 트라이 중 가장 좋은 트라이를 결정하는 일반적인 기법
  - 우선순위 큐를 사용하여 빈도수가 가장 작은 문자부터 차례로 트라이를 만듦
  - 인코딩된 메시지의 길이
    - 허프만 빈도수 트리의 가중치 외부 경로 길이 (weighted external path length)와 같게 됨
    - 동일한 빈도수를 가지는 경우
      - 허프만 트리가 최적해가 됨

허프만 트리의 구축 과정(1)

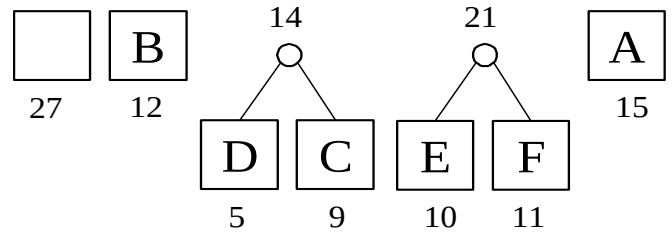
- 최소 히프
  - 허프만 트리의 자료구조에 가장 적합
- 주어진 텍스트의 빈도수를 계산

		A	B	C	D	E	F
k	0	1	2	3	4	5	6
count[k]	27	15	12	9	5	10	11

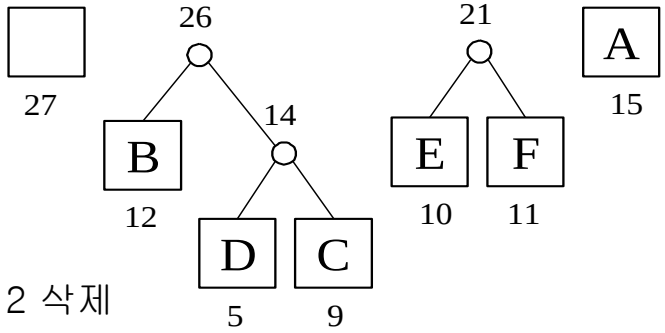
- 허프만 트리 구축



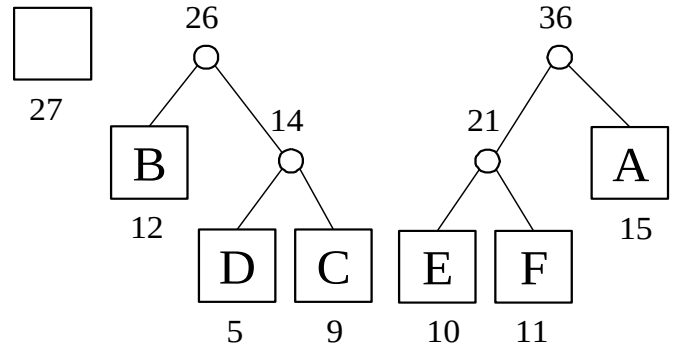
허프만 트리의 구축 과정(2)



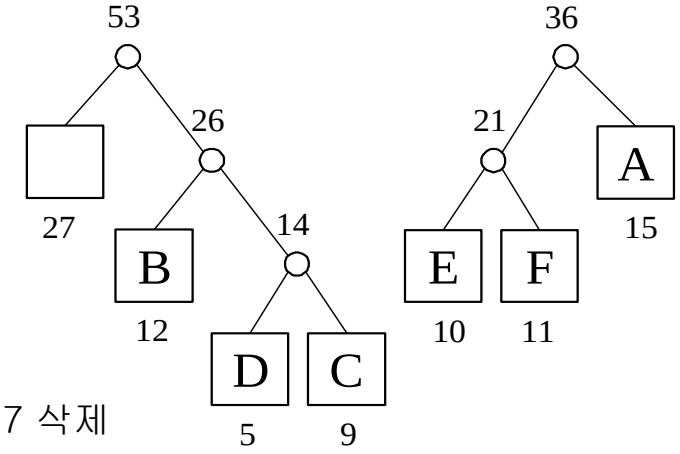
(c) 10과 11 삭제



(d) 14와 12 삭제

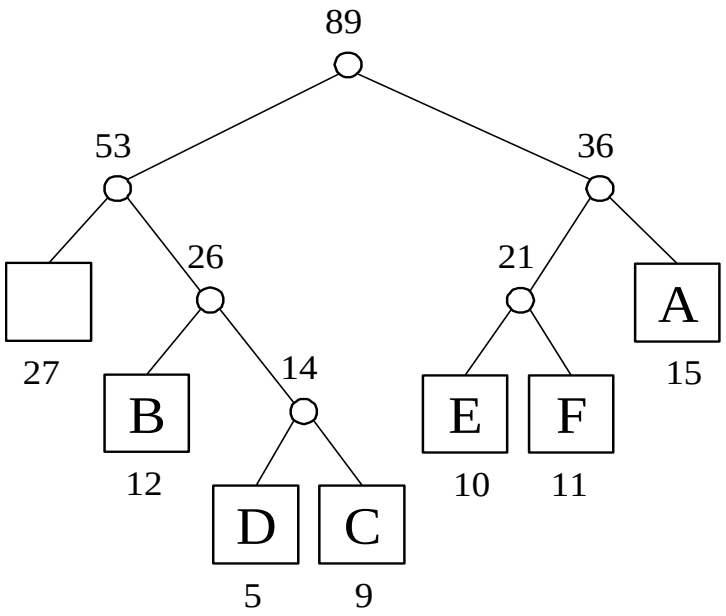


(e) 15와 21 삭제



(f) 26과 27 삭제

허프만 트리의 구축 과정(3)



(g) 36과 53 삭제

허프만 코드

- 허프만 인코딩에 사용될 코드를 할당

		A	B	C	D	E	F
k	0	1	2	3	4	5	6
len[k]	2	2	3	4	4	3	3
code[k]	0	3	2	7	6	4	5
	00	11	010	0111	0110	100	101

  - len[k] 허프만 코드의 길이
  - code[k] 십진수로 표현한 허프만 코드

허프만 인코딩 실행 예(1)

- VISION QUESTION ONION CAPTION GRADUATION EDUCATION
- 문자 빈도수

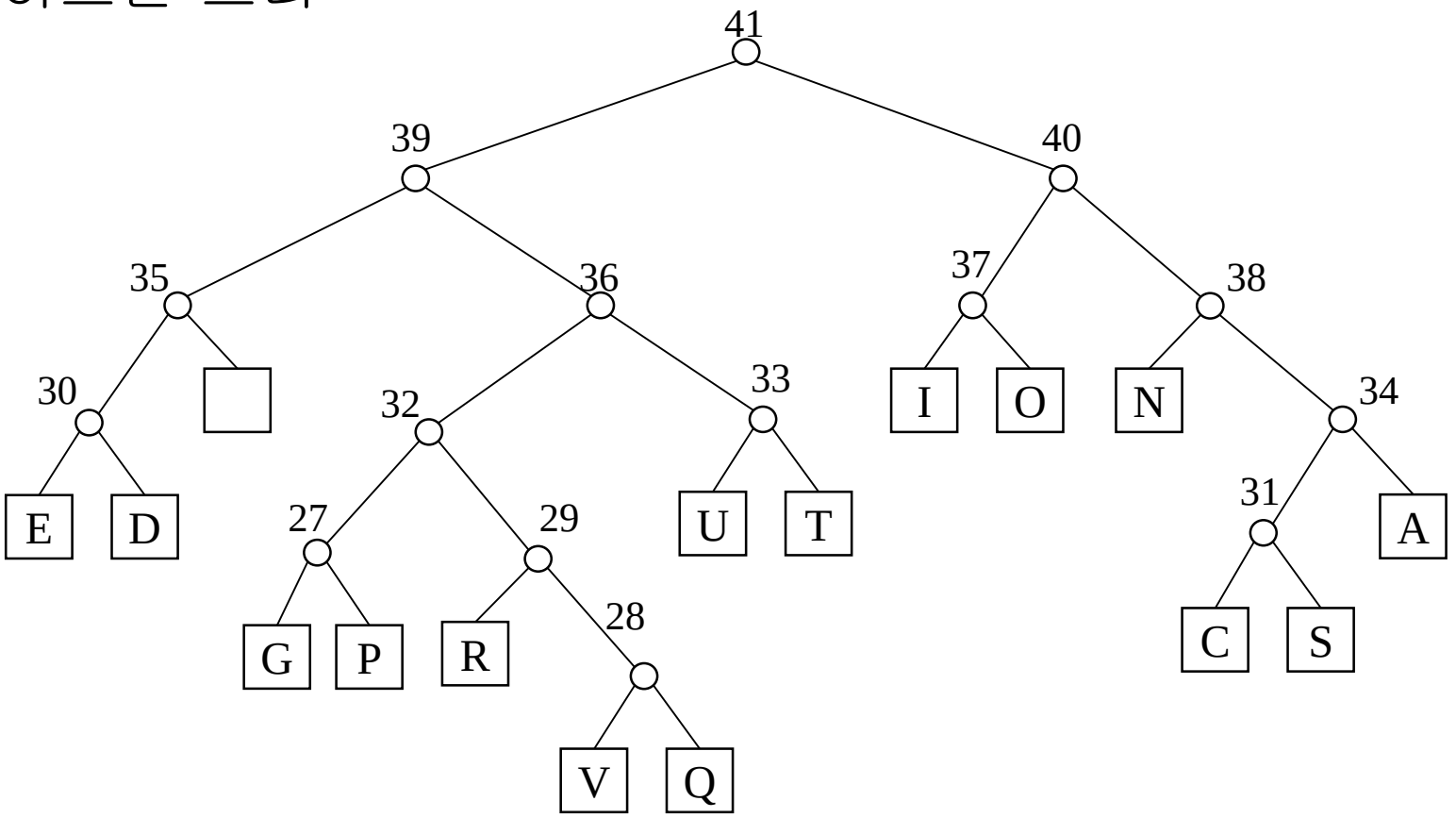
		A	C	D	E	G	I	N	O	P	Q	R	S	T	U	V
k	0	1	3	4	5	7	9	14	15	16	17	18	19	20	21	22
count[k]	5	4	2	2	2	1	7	7	7	1	1	1	2	4	3	1

- 부모 노드 표현

k	0	1	3	4	5	7	9	14	15	16	17	18	19	20	21	22
count[k]	5	4	2	2	2	1	7	7	7	1	1	1	2	4	3	1
dad[k]	-35	-34	31	-30	30	27	37	38	-37	-27	-28	29	-31	-33	33	28
k	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	
count[k]	2	2	3	4	4	5	7	8	9	12	14	15	21	29	50	
dad[k]	32	-29	-32	35	34	36	-36	-38	39	-39	40	-40	41	-41	0	

허프만 인코딩 실행 예(2)

◆ 허프만 트리



허프만 인코딩 실행 예(3)

◆ 허프만 코드

	k	code[k]	len[k]	
	0	1	3	001
A	1	15	4	1111
C	3	28	5	11100
D	4	1	4	0001
E	5	0	4	0000
G	7	8	5	01000
I	9	4	3	100
N	14	6	3	110
O	15	5	3	101
P	16	9	5	01001
Q	17	23	6	010111
R	18	10	5	01010
S	19	29	5	11101
T	20	7	4	0111
U	21	6	4	0110
V	22	22	6	010110

암호화 알고리즘

- 암호화(cryptology)
  - 일반적인 텍스트를 관련 없는 사람들이 읽을 수 없도록 하는 것
- 암호화 알고리즘
  - 컴퓨터를 사용하여 암호화를 수행할 때 사용되는 것
  - 암호 작성(cryptography)
    - 비밀 통신 시스템을 설계
  - 암호 해독(cryptanalysis)
    - 비밀 통신 시스템을 해독하는 방법을 연구

암호화 시스템

- 암호화 시스템(cryptosystem)
  - 송신자 : 메시지를 암호화(encrypt)
  - 수신자 : 메시지를 복호화(decrypt)
  - 암호화 기법(encryption method)
  - 복호화 기법(decryption method)
  - 키 매개변수(key parameters)

단순한 기법(1)

- 카이사르 암호화(Caesar cipher)
  - 고대 로마의 카이사르가 사용했다고 알려짐
  - 평문에 있는 문자가 알파벳의 N번째 문자라면, 이것을 (N+K)번째 문자로 교체
  - K=1일 때 카이사르 암호화의 예

평 문	S	A	V	E		P	R	I	V	A	T	E		R	Y	A	N
암호문	T	B	W	F	A	Q	S	J	W	B	U	F	A	S	Z	B	O

단순한 기법(2)

- 문자 변환표를 이용한 암호화
  - 단순한 카이사르 암호화보다 훨씬 강력한 암호화 기법
  - 문자 변환표의 예

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Q	H	C	B	E	J	K	A	R	W	S	T	U	V	D		I	O	P	X	Z	F	G	L	M	N	Y

  - 위 변환표를 사용한 암호화의 예

평 문	S	A	V	E		P	R	I	V	A	T	E		R	Y	A	N
암호문	X	H	G	J	Q	I	P	W	G	H	Z	J	Q	P	N	H	D

단순한 기법(3)

- 비즈네르(Vigenere) 암호화
  - 평문의 각 문자에 대해 서로 다른 변환표를 사용
  - 비즈네르 암호화의 예

키	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B
평 문	S	A	V	E		P	R	I	V	A	T	E		R	Y	A	N
암호문	T	C	Y	F	B	S	S	K	Y	B	V	H	A	T	A	B	P

- 버넘(Vernam) 암호화
  - 비즈네르 암호화에서 키의 길이를 평문의 길이와 같게 한 것
  - 유일하게 안전성이 증명된 암호화 시스템

단순한 기법(4)

- 이진수의 암호화
  - 배타적 논리합(exclusive-or) 연산 사용

x	XOR	*y*
0	0	0
0	1	1
1	1	0
1	0	1

암호화

키	1	1	0	0	0	1	1	1	0	0	1	1
평 문	1	0	0	1	1	0	0	0	1	1	1	0
암호문	0	1	0	1	1	1	1	1	1	1	0	1

복호화

키	1	1	0	0	0	1	1	1	0	0	1	1
암호문	0	1	0	1	1	1	1	1	1	1	0	1
평 문	1	0	0	1	1	0	0	0	1	1	1	0

단순한 기법(4)

- 암호화/복호화 장치
  (encryption/decryption machine)
  - 버넘 암호화 기법을 사용하기 어려운 많은 양의 데이터를 암호화할 수 있음
  - "의사 키(pseudo-key)" 사용
  - 키 스트림을 생성하는 과정
    - 해싱이나 난수 발생 알고리즘과 매우 유사

공개 키 암호화 시스템(1)

- 공개 키 암호화 시스템(public-key cryptosystem)
  - 안전하게 키를 분배하는 문제를 해결하기 위해 개발됨
  - 메시지를 보낼 때
    - 송신자는 수신자의 공개 키를 찾아 사용하여 암호화한 후 전송
  - 메시지를 읽을 때
    - 수신자는 자신이 가진 비밀 키(secret key)를 사용하여 복호화
  - 공개 키 암호화 시스템의 구성 조건
    - P 공개 키, S 비밀 키, M 메시지

    ① S(P(M)) = M

    ② 모든 (S, P) 쌍은 유일해야 한다.

    ③ P로부터 S를 알아내는 것은 M을 해독하는 것만큼 어려워야 한다.

    ④ S와 P는 쉽게 계산할 수 있어야 한다.

공개 키 암호화 시스템(2)

- 공개 키 암호화 시스템의 기본적인 전제
  - 주어진 큰 숫자가 소수인지를 결정하는 빠른 알고리즘은 있지만, 주어진 큰 비소수의 소수 인자를 찾아내는 빠른 알고리즘은 없음

    - 예 1)
      - 130자리 수가 소수인지 검사하는 데는 7분이 소요
      - 63자리 두 소수를 곱해서 얻은 수의 두 소수 인자를 찾아내는 데는 $4*10^6$년이 소요
    - 예 2)
      - 200자리 숫자가 어떤 두 소수의 곱으로 이루어지는지 알아내는 알고리즘은 수백만 년의 시간이 소요

RSA 알고리즘

- RSA (Rivest, Shamir and Adleman) 알고리즘
  - 공개 키 암호화 시스템에서 사용되는 대표적인 알고리즘
  - 메시지를 선형 시간에 암호화
  - 암호화 : $C = P(M) = M^P \bmod N$
  - 복호화 : $M = S(C) = C^S \bmod N$

RSA 알고리즘 사용 예(1)

- 공개키와 비밀키 계산 방법
  - 3 개의 소수를 선택
    - $s = 97$(비밀키: 가장 큰 수), $x = 47$, $y = 79$
  - N을 선택
    - $N = x \cdot y$ 이므로 $47 \cdot 79 = 3713$
  - p를 선택
    - $ps \bmod (x - 1)(y - 1) = 1$이 되는 p를 찾음
    - $p \cdot 97 \bmod 46 \cdot 78 = 1$
    - $p \cdot 97 \bmod 3588 = 1$
    - $p \cdot 97 = 3589$
    - $p = 3589 \div 97 = 37$
  - 공개 키 $p = 37$

RSA 알고리즘 사용 예(2)

- 암호화 과정
  - SAVE PRIVATE RYAN을 A는 01, B는 02, C는 03 등의 십진수로 인코딩
    - 19012205001618092201200500182501140 0
  - 암호화 계산
    - $1901^{37} \bmod 3713 = 0335$
    - $2205^{37} \bmod 3713 = 1472$
    - $0016^{37} \bmod 3713 = 1447$
    - $1809^{37} \bmod 3713 = 3060$
    - $2201^{37} \bmod 3713 = 1548$
    - $2005^{37} \bmod 3713 = 1608$
    - $0018^{37} \bmod 3713 = 3091$
    - $2501^{37} \bmod 3713 = 0654$
    - $1400^{37} \bmod 3713 = 1414$
  - 암호화된 메시지
    - 033514721447306015481608309106541414

RSA 알고리즘 사용 예(3)

- 복호화 과정
  - 복호화 계산
    - $0335^{97} \bmod 3713 = 1901$
    - $1472^{97} \bmod 3713 = 2205$
    - $1447^{97} \bmod 3713 = 0016$
    - …
  - 복호화된 메시지
    - 1901220500161809220120050018250114 00