string

≔ 태그

문자열

- 문자의 표현
- 문자열의 분류
- 문자열 비교
- 패턴매칭
- 문자열 암호화
  - brute_force , kmp, boyer_Moore

문자열

문자의 표현

- 확장 아스키는 7-bit를 사용하면서, 8-bit를 모두 사용함으로써 추가적인 문자를 표현할 수 있다.
- 표준 아스키는 마이크로컴퓨터 하드웨어 및 소프트웨어 사이에서 세계적으로 통용되는 것에 비해, 확장 아스키는 프로그램이나 컴퓨터 또는 프린터가 그것을 해독할 수 있도록 설계되어 있어야만 올바로 해독될 수 있다.
- 인터넷이 전세계적으로 발전하면서 아스키코드를 만들었을 때의 문제가 국가간에 정보를 주고 받을 때 발생했다.

- 자국의 코드체계를 타 국가가 가지고 있지 않으면 정보를 잘못해서 해석할 수 있다.

⇒ 따라서 다국어 처리를 위해 `유니코드` 라는 표준을 마련한 것이다.

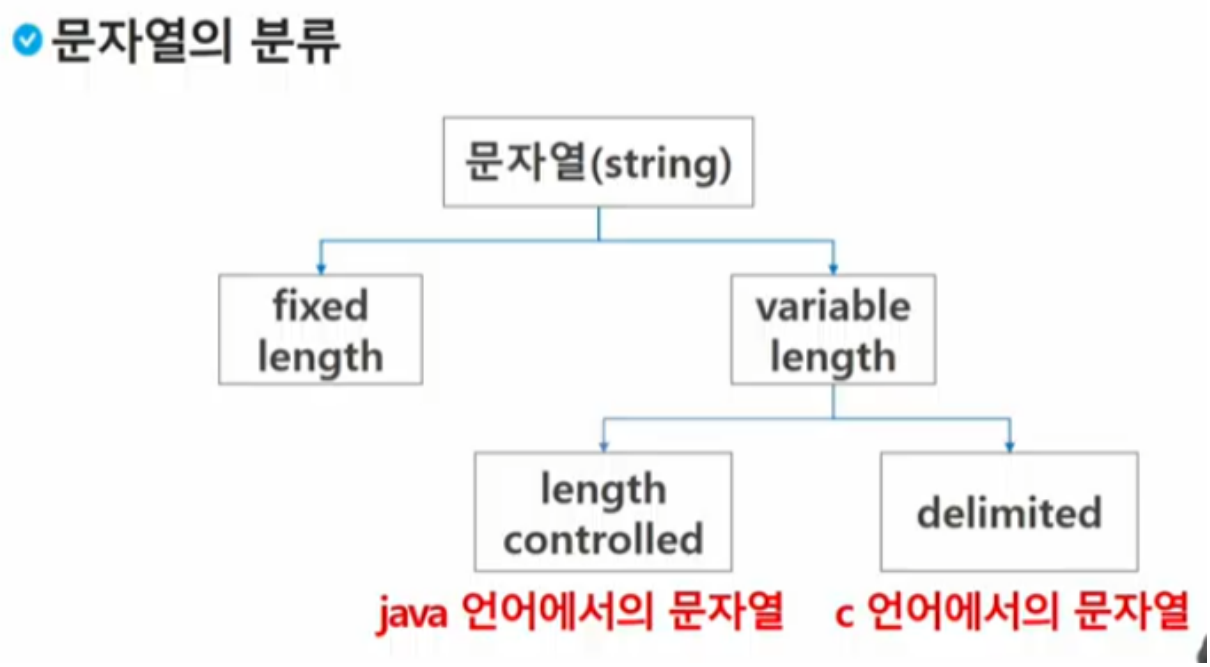
- 유니코드 인코딩

UTF-8 / MIN 8bit, MAX32bit (1 byte * 4) 유동

UTF-16 / MIN 16bit, MAX: 32bit (2 byte * 2) 유동

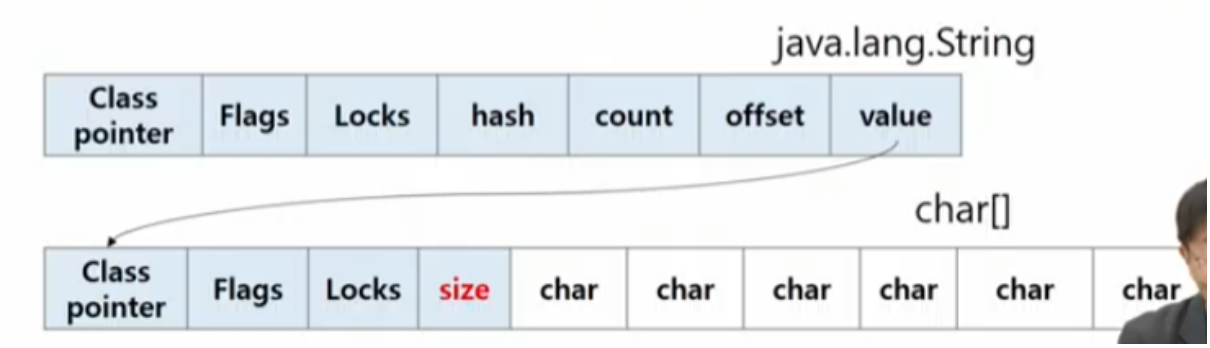
UTF-32 (in unix) / MIN 32bit, MAX: 32bit (4 byte * 1) 고정

문자열의 분류

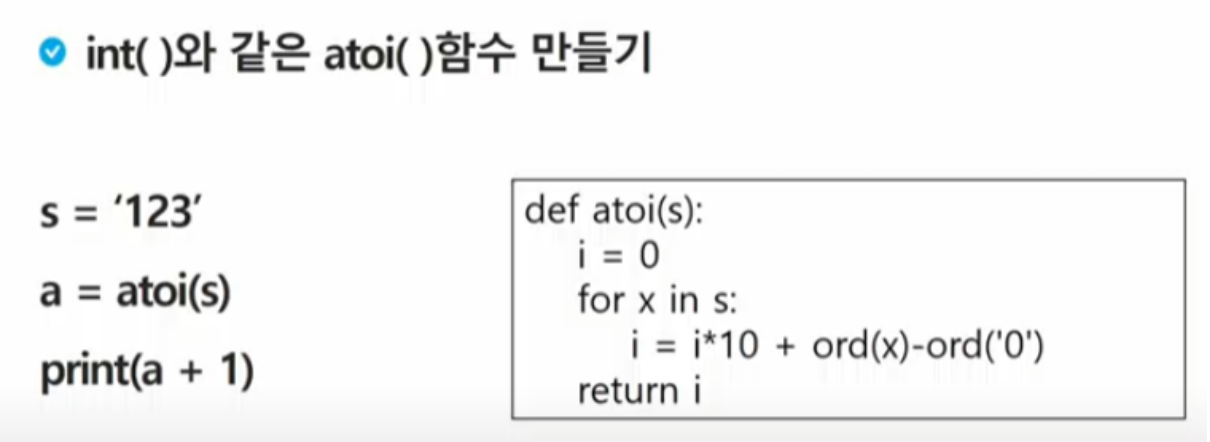


- java 에서 문자열 처리

hash 값, 문자열의 길이, 데이터의 시작점, 실제 문자열 배열에 대한 참조가 미리 메모리에 포함되어 있다.



문자열 비교



- 숫자를 문자열로 변환

```
def itoa(a):
    s = ''
    while a > 0:
        s = chr(a%10 + ord('0')) + s
        a //= 10
    return s
print(itoa(123))
print(type(itoa(123)))
```

- 아스키 코드 표

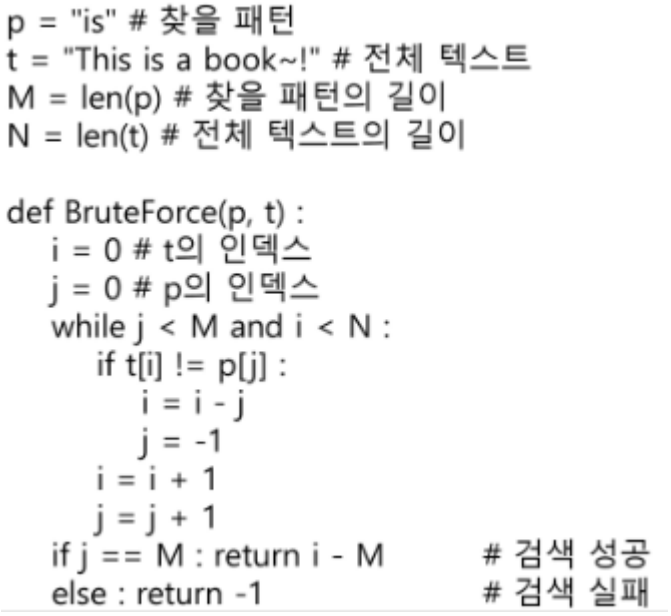
제어 문자		공백 문자		구두점		숫자		알파벳			
10진	16진	문자	10진	16진	문자	10진	16진	문자	10진	16진	문자
0	0x00	NUL	32	0x20	SP	64	0x40	@	96	0x60	`
1	0x01	SOH	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	39	0x27	'	71	0x47	G	103	0x67	g
8	0x08	BS	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	LF	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	VT	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	FF	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	CR	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E	SO	46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F	SI	47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10	DLE	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A	SUB	58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	ESC	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C	FS	60	0x3C	<	92	0x5C	₩	124	0x7C	|
29	0x1D	GS	61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E	RS	62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F	US	63	0x3F	?	95	0x5F	_	127	0x7F	DEL

패턴매칭

- 문자열을 검색하는 것.

1. **고지식한 패턴검색 알고리즘**

- 본문 문자열을 처음부터 끝까지 차례대로 순회하면서, 패턴 내의 문자들을 일일이 비교하는 방식으로 동작



```
p = "is" # 찾을 패턴
t = "This is a book~!" # 전체 텍스트
M = len(p) # 찾을 패턴의 길이
N = len(t) # 전체 텍스트의 길이

def BruteForce(p, t) :
    i = 0 # t의 인덱스
    j = 0 # p의 인덱스
    while j < M and i < N :
        if t[i] != p[j] :
            i = i - j
            j = -1
        i = i + 1
        j = j + 1
    if j == M : return i - M        # 검색 성공
    else : return -1                # 검색 실패
```

- 코드

```
def f(pat,txt,M,N):
    for i in range(N-M+1): # text에서 비교 시작 위치
        for j in range(M):
            if txt[i+j] != pat[j]: # 불일치면 다음 시작위치로
                break
        else: # 패턴 매칭에 성공하면
            return 1 # 1을 리턴한다.
```

```
        # 모든위치에서 비교가 끝난 경우
    return 0



T = int(input())
for tc in range(1,T+1):
    pat = input()
    txt = input()
    M   = len(pat)
    N   = len(txt)

    ans = f(pat, txt, M, N)
    print(f'#{tc} {ans}')
```

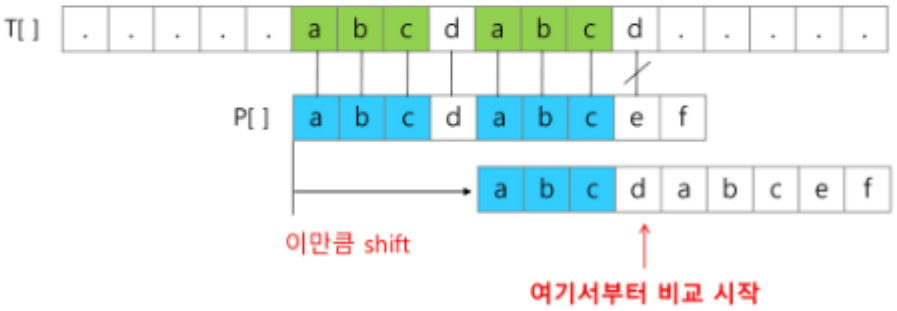
2. KMP 알고리즘

- 불일치가 발생한 텍스트 스트링의 앞 부분에 어떤 문자가 있는지를 알고 있으므로, 불일치가 발생한 앞 부분에 대해 다시 비교하지 않고 매칭 수행

- 패턴을 전처리하여 배열 next[m]을 구해서 잘못된 시작을 최소화

아이디어 설명

- 텍스트에서 abcdabc까지는 매치되고, e에서 실패한 상황 패턴의 맨 앞의 abc와 실패 직전의 abc는 동일함을 이용할 수 있다
- 실패한 텍스트 문자와 P[4]를 비교한다



- 코드

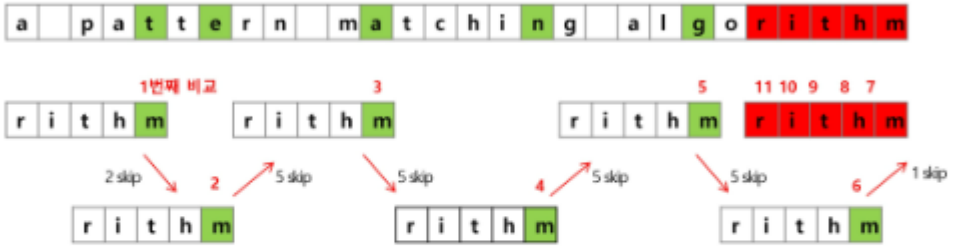
```
def kmp(t,p):
    N = len(t)
    M = len(p)
    lps = [0] * (M+1)
    # preprocessing
    j = 0
    lps[0] = -1
    for i in range(1,M):
        lps[i] = j # p[i] 이전에 일치한 개수
        if p[i] == p[j]:
            j += 1
        else:
            j = 0
    lps[M] = j
    # search
    while i < N and j <= M:
        if j == -1 or t[i] == p[j]:
            i += 1
            j += 1
        else:
            j = lps[j]
```

```
            if j == M:
                print(i-M, end = ' ')
        print()
        return
```

3. 보이어-무어 알고리즘

- 오른쪽에서 왼쪽으로 비교,
- 대부분의 상용 소프트웨어 채택하고 있는 알고리즘
- 보이어 무어 알고리즘은 오른쪽 끝에 있는 문자가 불일치 하고 이 문자가 패턴 내에 존재하지 않는 경우, 이동 거리는 무려 패턴의 길이 만큼이 된다.

보이어-무어 알고리즘을 이용한 예



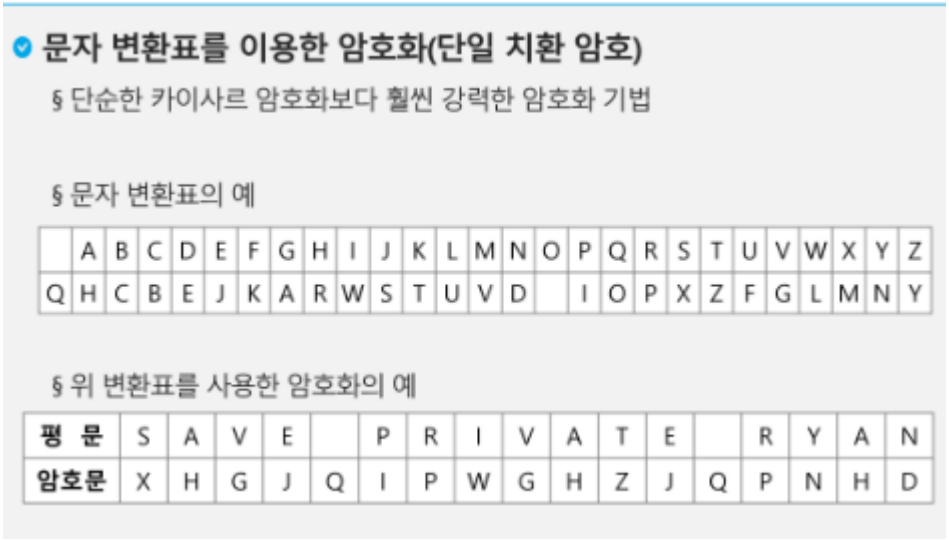
- rithm 문자열의 skip 배열

m	h	t	i	r	다른 모든 문자
0	1	2	3	4	5

문자열 암호화

- 시저 암호
  - 줄리어스 시저가 사용했다고 하는 암호이다
  - 시저는 기원전 100년경에 로마에서 활약했던 장군이다.

- 시저 암호에서는 평문에서 사용되고 있는 알파벳을 일정한 문자 수만큼 [평행이동] 시킴으로써 암호화를 진행한다.



문자 변환표를 이용한 암호화(단일 치환 암호)

§ 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

- bit 열의 암호화

bit열의 암호화

§ 배타적 논리합(exclusive-or) 연산 사용

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

암호화

평 문	1	0	0	1	1	0	0	0	1	1	1	0
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

해 독

암호문	0	1	0	1	1	1	1	1	1	1	0	1
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

brute_force , kmp, boyer_Moore

- 글자수 세기

1. brute_force

```
def brute_force(pattern, target):
    # 둘다 첫 조사 시작지점 0번에서 시작
    pattern_index = 0
    target_index  = 0
    # 현재 조사 위치가 조사대상의 범위를 벗어나기 전까지
    while target_index < len(target):
        # 일치하지 않으면
        if pattern[pattern_index] != target[target_index]:
            pattern_index = -1
        # 일치하면 => 사실상항상
        target_index += 1
        pattern_index += 1
        # 패턴의 끝까지 index가 증가했다
        # -> target과 pattern이 일치하지 않는 부분 없이
        # 패턴의 끝까지 조사했다.
        if pattern_index == len(pattern):
            return True
    return False
```

2. kmp

```
def KMP(pattern,target):
    def make_lps():
        # 내앞에 나와 동일한 패턴이 몇번 나왔는지 세어주는 리스트
        lps = [0] * len(pattern)
        for idx in range(1,len(pattern)): # 번 인덱스는 앞에 중복
            if pattern[lps[idx-1]] == pattern[idx]:
                lps[idx] = lps[idx - 1] + 1
        lps.insert(0,-1)
```

```
        return lps

    lps = make_lps()

    pattern_index = 0
    target_index  = 0
    # 현재 조사 위치가 조사대상의 범위를 벗어나기 전까지
    print(lps)
    while target_index < len(target):
        print(lps[pattern_index])
        print(target_index, target[target_index], pattern_ind
        # 일치하지 않으면
        if pattern[pattern_index] != target[target_index]:
            pattern_index = lps[pattern_index]
        # 일치하면 => 사실상항상
        target_index += 1
        pattern_index += 1
        # 패턴의 끝까지 index가 증가했다
        # -> target과 pattern이 일치하지 않는 부분 없이
        # 패턴의 끝까지 조사했다.
        if pattern_index == len(pattern):
            return True
    return False

T = int(input())
for tc in range(1,T+1):
    str1 = input()
    str2 = input()
    print(KMP(str1, str2))
```

3. boyre_moore

```
def boyer_moore(pattern, target):
    lps = {pattern[idx] : len(pattern) - 1 - idx for idx in r
    pattern_idx = len(pattern)
    target_idx  = 0
```

```
    while target_idx <= len(target) - pattern_idx:
        for p_idx in range(pattern_idx-1,-1,-1):
            if target[target_idx + p_idx] != pattern[p_idx]:
                target_idx += lps.get(target[target_idx + p_i
                break # 틀렸으니까 p_idx 다시 len(patter - 1) 까지
        else:
            return True
    return False
```