# 19장 비트 조작

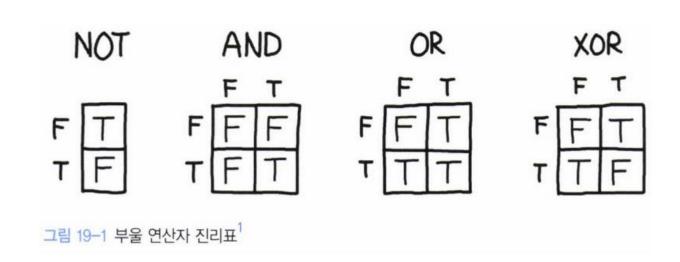
투빅스 11기 김대웅

## Intro

- Logic Gate Claude Shannon(1937)
  - Boolean Algebra를 회로에 적용
- Logic Circuit
  - 현대 모든 디지털 컴퓨터의 기본 개념이자 근간
  - 하드웨어 뿐만 아니라 다양한 부분에 널리 활용 → 프로그래밍에서의 활용이 이번 장의 내용

# **Boolean Operation**

- 기본 부울 연산자
  - NOT
  - AND
  - OR
- 보조 부울 연산자
  - XOR[EOR;EXOR]



# **Boolean Operation**

- 기본 부울 연산자
  - NOT
  - AND
  - OR
- 보조 부울 연산자
  - XOR

### NOT

not True

False

### OR

True or False

True

### **AND**

True and False

False

### XOR

x = y = True

(x and not y) or (not x and y)

False

# Bitwise Operator(비트 연산자)

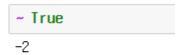
• Bitwise NOT : ~

• Bitwise OR :

• Bitwise AND : &

• Bitwise XOR : ^

### **Bitwise NOT**



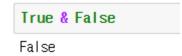
### **Bitwise OR**



### Bitwise XOR

True ^	True	
False		

### **Bitwise AND**



# 파이썬 진법 표현

- bin(십진수(int)) → 이진수(str)
  - e.g.)  $bin(87) \rightarrow '0b1010111'$
- Int(이진수(str), 2) → 십진수(int)
  - e.g.) int('0b10111', 2) → 87
- b = 이진수(not str) → b를 십진수(int)로 처리

```
• e.g.) >>> b = 0b1010111
>>> b
87
>>> type(b)
<class 'int'>
```

- 산술 연산 & 비트 연산
  - 덧셈&곱셈: 십진수와 동일

### **ADD**

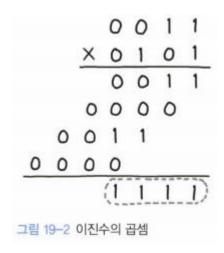
bin(0b0110 + 0b0010)
'Ob1000'

### MUL

bin(0b0011 \* 0b0101)
'Ob1111'

- 곱셈과 시프팅
  - 기존 입력값 0011을 이후 입력값의 자리수만큼 시프팅한 결과의 합
  - E.g. 첫 번째와 세번째 자리가 1 → 첫 번째 결과는 동일한 0011 세 번째 결과는 두 자리 시프팅한 001100





- 시프팅 연산자
  - >> #
    - 오른쪽으로 #칸 시프팅
    - 십진수 값으로 2배씩 감소
    - 우측이 잘려나감
  - << #
    - 왼쪽으로 #칸 시프팅
    - 십진수 값으로 2배씩 증가
    - 곱셈연산에 이용

### Shifting - 1

```
# to Right
bin(Ob1101 >> 2)
```

'Ob11'

### Shifting - 2

```
# to Left
bin(0b1101 << 2)
```

'0b110100'

- 계산 과정
  - 1. Bitwise NOT : ~0b1100 → 0b0011
  - 2. Bitwise XOR: 0b0101 ^ 0b0011 = 0b0110
- 0b0110(10진수 6)이 나와야 하지만 결과는 -0b1010 (10진수 -10)

```
bin(0b0101 ^ ~0b1100)
'-0b1010'
```

- 결과값이 달라진 이유
  - Bitwise NOT = 2의 보수 1
  - 0b1100 (10진수 12)의 Bitwise NOT을 적용한 결과는 10진수로 표현하면 x = x 1 따라서 -12 1 = -13
  - -13을 2의 보수로 표현 → 111111...110011
  - 앞의 값 0b0101 → 000000...000101
  - Bitwise XOR 결과 → 111111...110110
  - 2의 보수로 표현 → -10

### • 참고

- 2의 보수 : 십진수의 음수를 이진수의 음수로 표현한 것. 십진수의 이진법 표현을 반전 시킨 값(1의 보수) + 1
- Bitwise NOT = 2의 보수 1 인 이유
  - 2의 보수 = 1의 보수 + 1
  - 1의 보수 = 각 비트의 자리를 반전 → Bitwise NOT

# 자릿수 제한 비트 연산

- 결과값이 달라진 이유
  - Bitwise NOT
    - 00000...001100 (10진수 12) → 111111...110011 (10진수 -13)
  - Bitwise XOR(실제 결과)
    - 00000...000101 XOR 111111...110011 → 111111...110110 (10진수 -10)
  - Bitwise XOR(의도한 결과)
    - 00000...000101 XOR 000000...000011 → 000000...000110 (10진수 6)
- → Bitwise NOT에서 앞부분에 대한 MASK 처리, 자릿수 제한

# 자릿수 제한 비트 연산

- 비트 마스크와 XOR을 이용한 Bitwise NOT 대체
  - 0b1100 → 0b0011 로 바꾸기 위해
    Bitwise NOT 연산자를 그대로 사용하는 대신
    자릿수 만큼의 최댓값을 지닌 비트 마스크 MASK를 만들어
    그 값과 XOR을 통해 값을 만든다.

```
bin(0b1100 & 0b1111)

'Ob1100'

MASK = Ob1111
bin(0b0101 ^ (0b1100 ^ MASK))

'Ob110'
```

# 2의 보수(Two's Complement)

- 2의 보수 숫자 포맷
  - 음수를 저장하기 위해 사용하는 방법 중 하나
  - E.g. 4비트 숫자 표현
    - 부호 비트(MSB, Most Significant Bit; 최상위비트)
    - 양수 0xxx, 음수 1xxx
    - 표현 범위 : -2^(n-1) ~ 2^(n-1) -1 (n=4이면 -8 ~ 7)
- 4비트로 2의 보수 표현
  - 비트 마스크를 이용해 자릿수 제한하여 계산
  - MASK = 0xF  $\rightarrow 000000...001111$
  - bin(-8) → 111111...111000
  - Bin(-8&MASK) → 000000...001000

표 19-1 4비트 정수 2의 보수

십진수	2의 보수	
0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
-8	1000	
-7	1001	
-6	1010	
-5	1011	
-4	1100	
-3	1101	
-2	1110	
-1	1111	

# 71. 해밍 거리

두 정수를 입력받아 몇 비트가 다른지 계산하라.

• 입력

$$x = 1, y = 4$$

• 출력

2

- 설명
  - 1 (0 0 0 1)
  - 4 (0 1 0 0)

T

화살표 표시한 두 군데 비트가 다르므로 정답은 2다.

# 71. Hamming Distance(해밍 거리)

- 풀이
  - 두 이진수의 다른 위치의 수 → XOR 연산 후 1의 개수 카운트

```
class Solution:
def hammingDistance(self, x: int, y: int) -> int:
return sum(map(lambda x : int(x), bin(x ^ y)[2:]))
```

```
class Solution:
def hammingDistance(self, x: int, y: int) -> int:
return bin(x ^ y).count('1')
```

# 74. 1비트의 개수

부호없는 정수형Unsigned Integer을 입력받아 1비트의 개수를 출력하라.

### 예제 1

입력

#### 

• 출력

3

### 예제 2

입력

#### 

출력

1

### 예제 3

• 입력

#### 

출력

31

# 74. 1비트의 개수

- 풀이
  - 1의 개수 카운트

- 비트 연산을 이용한 방법
  - 1을 뺀 값과 자기자신의 AND 연산 → 이진수에서 마지막 비트를 제거
  - E.g. 1010 (10진수 10), 1000 (10진수 9) 일 때 둘의 AND 연산 → 1000
  - 뒤에서부터 비트를 제거한 횟수

```
class Solution:
def hammingWeight(self, n: int) -> int:
return bin(n).count('1')
```

```
1 class Solution:
2 def hammingWeight(self, n: int) -> int:
3 count = 0
4 while n:
5 # 1을 뺀 값과 AND 연산 횟수 측정
6 n &= n - 1
7 count += 1
8 return count
```