비트마스크

광운대학교
소프트웨어학부

김석희

비트

> 비트(bit)

비트는 이진수(binary digit)로, 컴퓨터에서 사용되는 데이터의 최소 단위이다.

0 / 1 두 개의 값만을 가질 수 있으며, 이 두가지로 숫자를 표현하는 방법을 이진법이라고 한다.

비트마스크

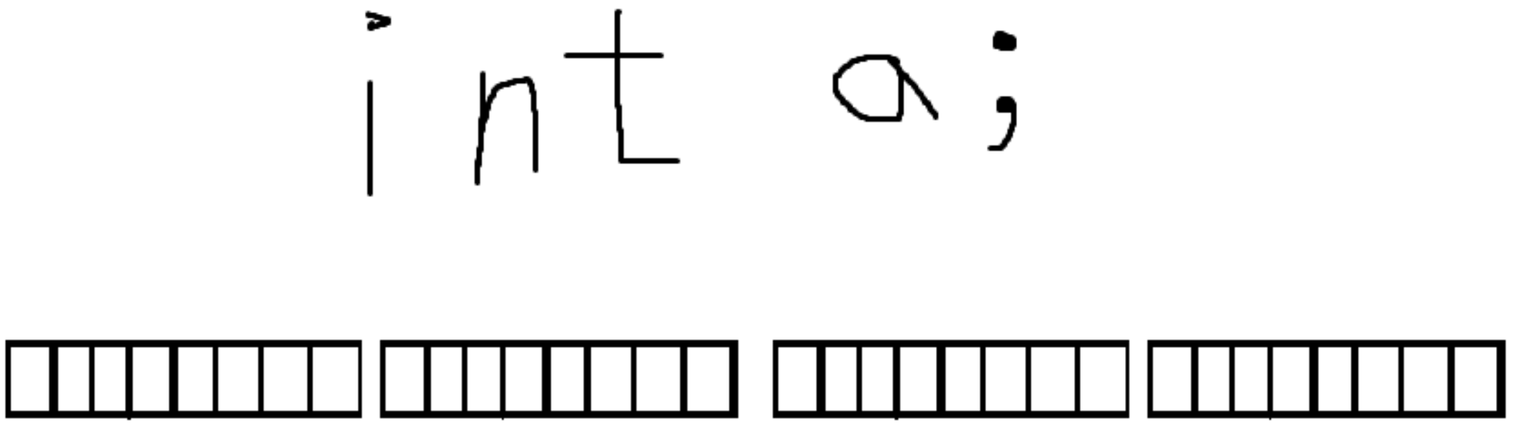
1. 비트마스크(BitMask)란?

- 비트마스크(BitMask)는 이진수를 사용하는 컴퓨터의 연산 방식을 이용하여, **정수의 이진수 표현을 자료 구조로 쓰는 기법**을 말한다.

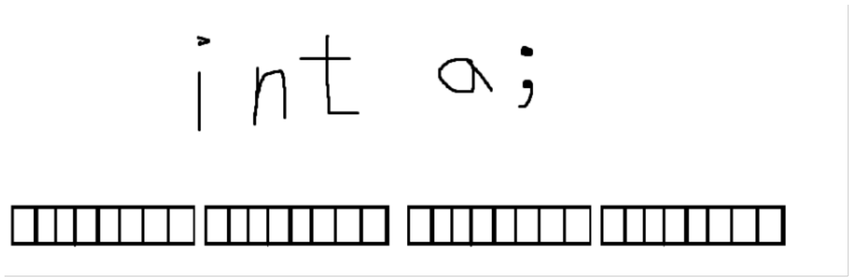
- 이진수는 0 또는 1을 이용하므로 하나의 비트(bit)가 표현할 수 있는 경우는 두 가지이다.

- 보통 어떤 비트가 1이면 "켜져 있다"라고 말하며, 0이면 "꺼져 있다"라고 말한다.

비트마스크



비트마스크



1 1 0 1 (2진수) → 13 (10진수)

2^3 2^2 2^1 2^0

(8 * 1) + (4 * 1) + (2 * 0) + (1 * 1) = 13

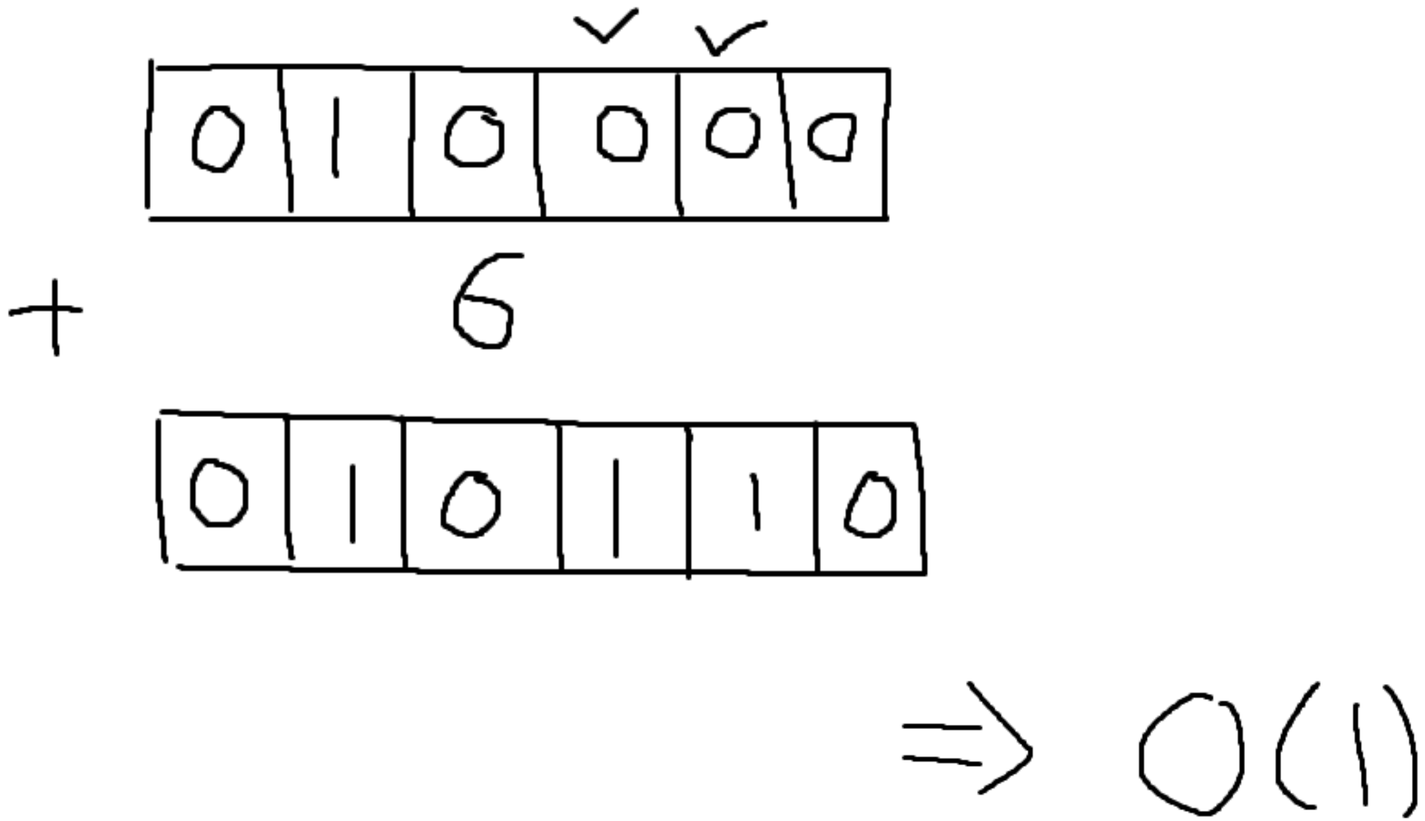
비트마스크의 장점

비트마스크는 크게 어려운 개념이 아니며, 이 개념을 알고 있다면 매우 유용한 경우가 꽤나 있다. 비트마스크의 장점들은 다음과 같다.

1. 수행 시간이 빠르다.

비트마스크 연산은 bit 연산이기 때문에 **O(1)**에 구현되는 것이 많다. 따라서 다른 자료구조를 이용하는 것보다 훨씬 빠르게 동작하게 된다.
다만, 비트마스크를 이용하는 경우에는 비트의 개수만큼 원소를 다룰 수 있기 때문에 연산 횟수가 적은 경우에는 속도에 큰 차이가 없지만, 연산 횟수가 늘어날수록 차이가 매우 커지게 된다.

비트마스크의 장점



비트연산 되짚어 보기

시프트(Shift) 연산(>>, <<)

왼쪽 또는 오른쪽으로 비트를 옮긴다.

```
00001010 << 2 = 101000
00001010 >> 2 = 000010
```

비트마스크의 장점

2. 코드가 짧다.

다양한 집합 연산들을 비트연산자로 한 줄로 작성할 수 있기 때문에 반복문, 조건문 등을 이용한 코드보다 훨씬 간결한 코드를 작성할 수 있다.

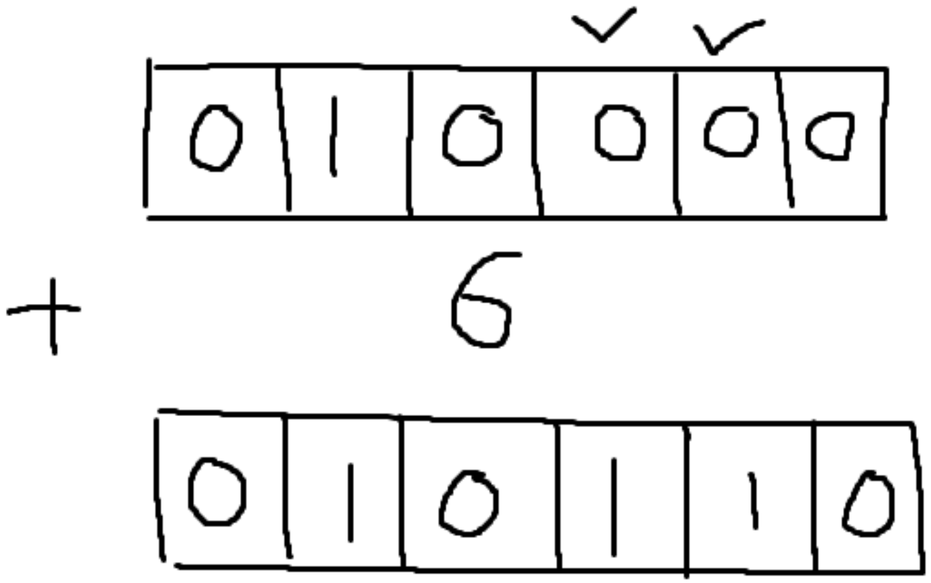
```
void solve(){
    for(int i = 0 ; i < N; i++){
        if(!(M & (1 << i))){
            answer = "OFF";
            return;
        }
    }
    answer = "ON";
}
```

비트마스크의 장점

3. 메모리 사용량이 더 적다.

개인적으로, 비트마스크를 이용하는 가장 큰 이유라고 생각한다.
간단한 예시로, bit가 10개인 경우에는 각 bit당 두 가지 경우를 가지기 때문에 2^{10}가지의 경우를 10bit 이진수 하나로 표현이 가능하다.
이처럼 하나의 정수로 매우 많은 경우의 수를 표현할 수 있기 때문에 메모리 측면에서 효율적이며, 더 많은 데이터를 미리 계산해서 저장해 둘 수 있는 장점이 있다. (*DP에 매우 유용하다*)

비트마스크의 장점




```
int[] array1 = [1, 1, 1, 1, 0];
int[] array2 = [1, 1, 0, 1, 0];
int[] array3 = [1, 0, 1, 0, 0];
```

int a

비트연산 되짚어 보기

NOT, AND, OR, XOR

A	B	~A	A & B	A | B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

비트연산 되짚어 보기

AND 연산(&)

대응하는 두 비트가 모두 1일 때, 1을 반환.

```
1010 & 1111 = 1010
```

비트연산 되짚어 보기

OR 연산(|)

대응하는 두 비트가 모두 1 또는 하나라도 1일 때, 1을 반환.

```
1010 | 1111 = 1111
```

비트연산 되짚어 보기

XOR 연산(^)

대응하는 두 비트가 서로 다르면 1을 반환.

```
1010 | 1111 = 0101
```

비트연산 되짚어 보기

NOT 연산(~)

비트의 값을 반전하여 반환.

```
~1010 = 0101
```

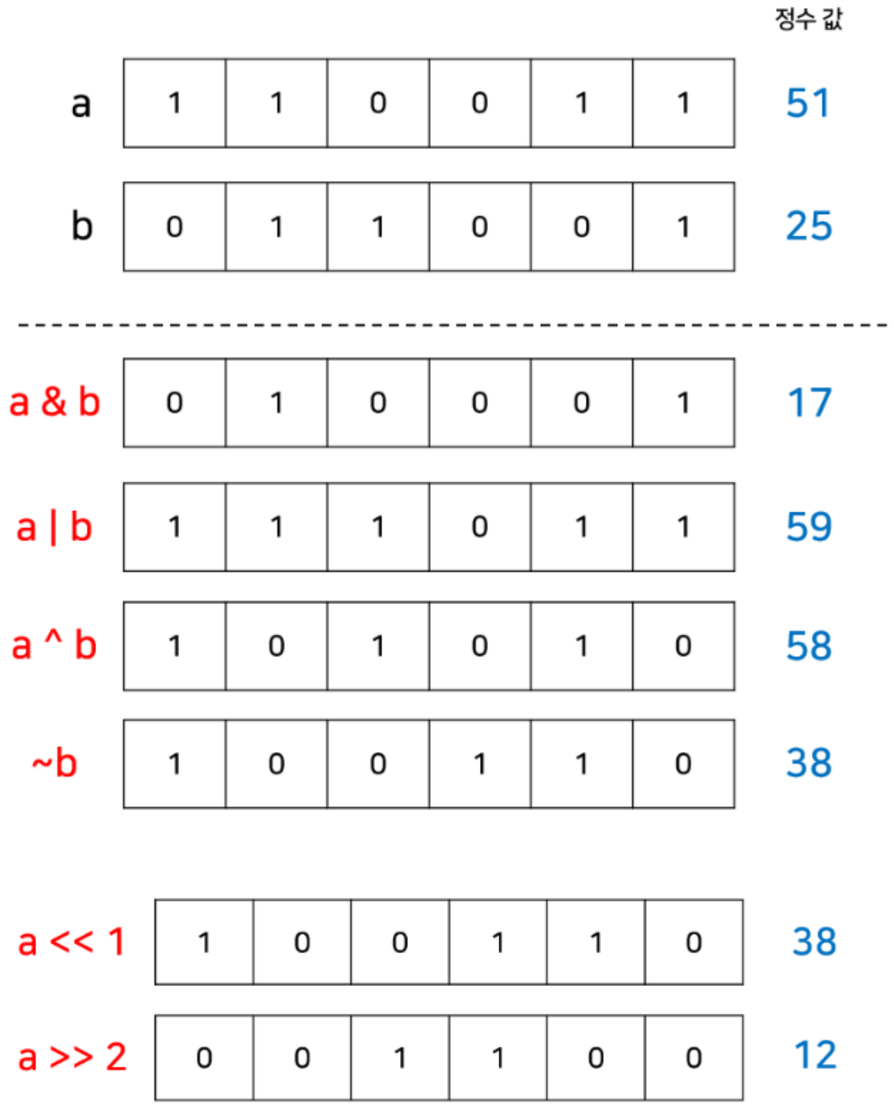
비트연산 되짚어 보기

시프트(Shift) 연산(>>, <<)

왼쪽 또는 오른쪽으로 비트를 옮긴다.

```
00001010 << 2 = 101000
00001010 >> 2 = 000010
```

비트연산 되짚어 보기


정수 값
a
1 1 0 0 1 1
51
b
0 1 1 0 0 1
25
a & b
0 1 0 0 0 1
17
a | b
1 1 1 0 1 1
59
a ^ b
1 0 1 0 1 0
58
~b
1 0 0 1 1 0
38
a << 1
1 0 0 1 1 0
38
a >> 2
0 0 1 1 0 0
12

예시

```cpp
#include <iostream>
#include <stdio.h>

// i& ( 1 << j ) i의 j번째 비트가 1인지 아닌지를 의미한다.
void printSubsets(char arr[], int n){
    for (int i = 0; i < (1 << n); ++i){
        printf("{");
        for (int j = 0; j<n; ++j){
            if (i & (1 << j))
                printf("%c ", arr[j]);
        }
        printf("}\n");
    }
}

int main(void){
    char data[] = {'A', 'B', 'C', 'D'};
    printSubsets(data, 4);
    return 0;
}
```

```
{}
{A }
{B }
{A B }
{C }
{A C }
{B C }
{A B C }
{D }
{A D }
{B D }
{A B D }
{C D }
{A C D }
{B C D }
{A B C D }
```

예시

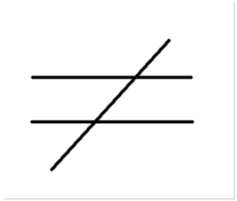
```cpp
#include <iostream>
#include <stdio.h>

// i& ( 1 << j ) i의 j번째 비트가 1인지 아닌지를 의미한다.
void printBits(char n){
    for (int i = 7; i >= 0; --i){
        if( n & (1 << i)) printf("1");
        else printf("0");
    }
}

int main(int argc, char** argv)
{
    char i;
    for (i = -5; i < 6; ++i){
        printf("%3d = ", i);
        printBits(i);
        printf("\n");
    }
    return 0;
}
```

```
-5 = 11111011
-4 = 11111100
-3 = 11111101
-2 = 11111110
-1 = 11111111
 0 = 00000000
 1 = 00000001
 2 = 00000010
 3 = 00000011
 4 = 00000100
 5 = 00000101
```

번외) 보수

-1  1000000.....0000000001

-1  1111111.....1111111111

번외) 보수

컴퓨터에서 보수가 필요한 이유

컴퓨터는 이진수만 이해할 수 있습니다. 그렇다면 음수는 어떻게 표현할 수 있을까요? 음수를 표현하기 위해 보수가 사용됩니다. 컴퓨터에서 보수는 양수를 음수화로 표현하기 위한 방법이 됩니다. 즉, 보충하는 수. 양수에 대한 보수가 음수입니다. 컴퓨터의 CPU는 뺄셈도 모두 덧셈으로 처리한다는 것을 알고 계실 겁니다. 보수를 사용하게 되면 5-5=0의 과정을 컴퓨터는 5+(-5)=0으로 표현되게 되는것입니다.

번외) 보수

2^3	2^2	2^1	2^0
0	0	0	1

✚

2^3	2^2	2^1	2^0
1	1	1	1

=

2^4	2^3	2^2	2^1	2^0
1	0	0	0	0

위의 그림을 보시면 이해가 되실 겁니다. 0001_2 + 와 그의 보수인 1111_2 을 더했더니 10000_2 이 된 것을 보실 수 있으실 겁니다. 여기서 최상위 비트인 1을 날려버리게 되면 0이 되는 것입니다.

번외) 보수

2의 보수

10진수	2진수	음수	부호 절대값 방식	1의 보수	2의 보수
0	0000	-0	1000	1111	0000
1	0001	-1	1001	1110	1111
2	0010	-2	1010	1101	1110
3	0011	-3	1011	1100	1101
4	0100	-4	1100	1011	1100
5	0101	-5	1101	1010	1011
6	0110	-6	1110	1001	1010
7	0111	-7	0111	1000	1001
8	1000	-8	-	-	1000

번외) 보수

보수 계산법

1의 보수 구하기

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	0	0	0	0	1	1	1	**00000111_2 (7)**
1	1	1	1	1	0	0	0	**00000111_2의 1의 보수**

십진수로는 7이고 이진수로는 00000111_2 의 보수를 구하는 과정입니다. 1의 보수를 구하는 방법은 굉장히 간단합니다. 1을 0으로 0을 1로 NOT연산을 해주시면 됩니다.

2의 보수 구하기

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
1	1	1	1	1	0	0	0	
1	1	1	1	1	0	0	1	**+1**

2의 보수를 구하려면 앞서 구했던 1의 보수에 1을 더해주시면 됩니다.

번외) 보수

2의 보수

10진수	2진수	음수	부호 절대값 방식	1의 보수	2의 보수
0	0000	-0	1000	1111	0000
1	0001	-1	1001	1110	1111
2	0010	-2	1010	1101	1110
3	0011	-3	1011	1100	1101
4	0100	-4	1100	1011	1100
5	0101	-5	1101	1010	1011
6	0110	-6	1110	1001	1010
7	0111	-7	0111	1000	1001
8	1000	-8	-	-	1000

비트마스크 연산

1. S에 X가 있는지 검사

= S의 X번째 비트가 1인지 0인지 검사

= X번째 비트만 1로 두고 AND 연산 수행

= `S & (1 << X)`

비트마스크 연산

2. S에 X를 추가

= S의 X번째 비트를 1로 변경

= X번째 비트만 1로 두고 OR 연산 수행

= `S | (1 << X)`

비트마스크 연산

3. S에서 X를 삭제

= S의 X번째 비트를 0으로 변경

= X번째 비트만 0으로 두고 AND 연산 수행

= `S & ~(1 << X)`

비트마스크 연산

4. S에서 X를 토글

= S의 X번째 비트가 0이면 1로, 1이면 0으로 변경

= X번째 비트만 1로 두고 XOR 연산 수행

= `S ^ (1 << X)`

비트마스크 연산

1	뜻	비트를 왼쪽으로 n만큼 이동(2^n)	(1 << n)
	예시	// (1 << 0) == 1 비트를 4만큼 이동하면 10000	

비트마스크 연산

2	뜻	n번째 비트 켜기	num |= (1 << n)
	예시	10001의 2번째 비트를 키면 10101	

비트마스크 연산

3	뜻	n번째 비트 끄기	num &= ~(1 << n)
	예시	10111의 2번째 비트를 끄면 10011	

비트마스크 연산

4	뜻	n번째 비트가 켜져있으면 True, 아니면 False	if(num & (1 << n))
	예시	10101의 2번째 피트가 켜져 있으므로 True	

비트마스크 연산

5	뜻	n번째를 비트를 반전 (XOR)	num ^= (1 << n)
	예시	10101의 2번째를 반전하면 10001	

비트마스크 연산

7	뜻	켜져 있는 비트 중에서 마지막 비트 추출	n & -n
	예시	1011100의 마지막 비트를 추출하면 100	

비트마스크 연산

8	뜻	아래부터 꺼진 비트 켜기	n | (n+1)
	예시	0100 -> 0101 -> 0111 -> 1111	
9	뜻	아래부터 켜진 비트 끄기	n & (n-1)
	예시	0111 -> 0110 -> 0100 -> 0000	

11723

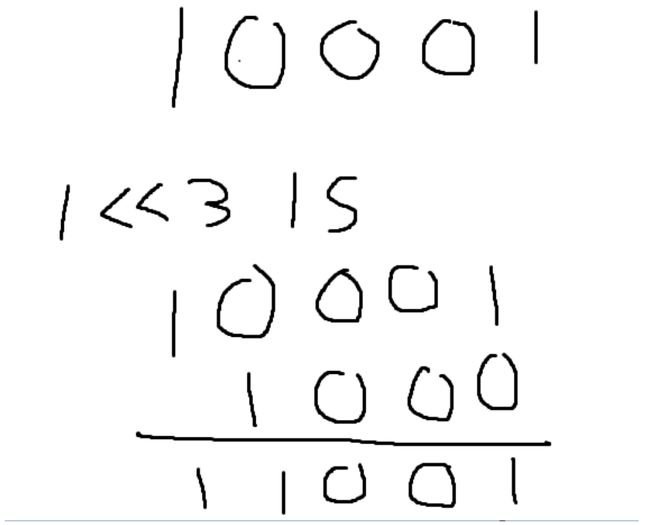
비어있는 공집합 S가 주어졌을 때, 아래 연산을 수행하는 프로그램을 작성하시오.

- `add x` : S에 x를 추가한다. (1 ≤ x ≤ 20) S에 x가 이미 있는 경우에는 연산을 무시한다.
- `remove x` : S에서 x를 제거한다. (1 ≤ x ≤ 20) S에 x가 없는 경우에는 연산을 무시한다.
- `check x` : S에 x가 있으면 1을, 없으면 0을 출력한다. (1 ≤ x ≤ 20)
- `toggle x` : S에 x가 있으면 x를 제거하고, 없으면 x를 추가한다. (1 ≤ x ≤ 20)
- `all` : S를 {1, 2, ..., 20} 으로 바꾼다.
- `empty` : S를 공집합으로 바꾼다.

11723

- add x : S에 x를 추가한다. (1 ≤ x ≤ 20) S에 x가 이미 있는 경우에는 연산을 무시한다.

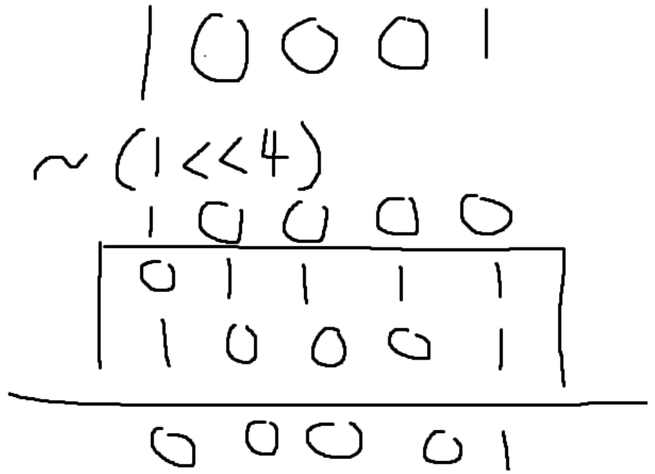
```
if (cmd == "add") {
    cin >> x;
    x--;
    s = (s | (1 << x));
```



11723

- remove x : S에서 x를 제거한다. (1 ≤ x ≤ 20) S에 x가 없는 경우에는 연산을 무시한다.

```
} else if (cmd == "remove") {
    cin >> x;
    x--;
    s = (s & ~(1 << x));
```



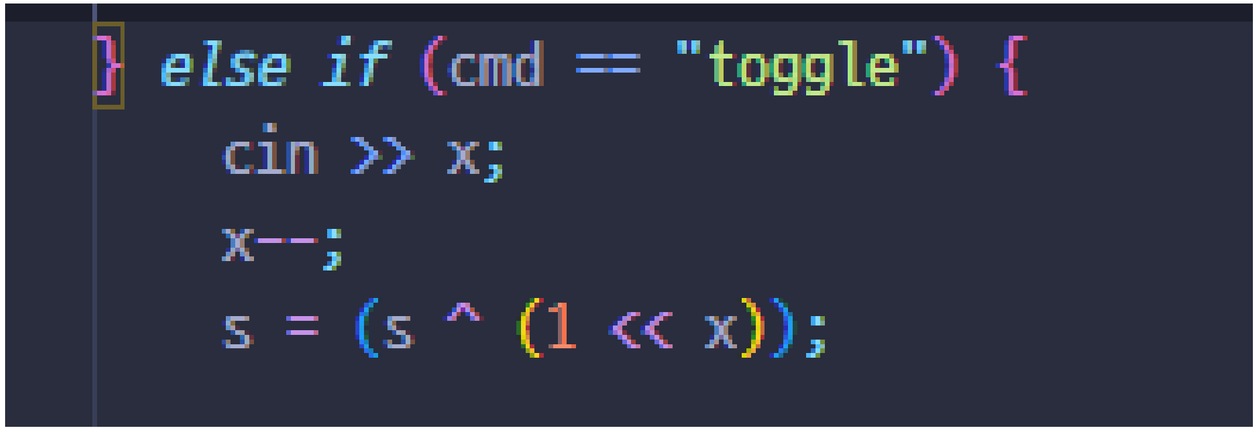
11723

- check x : S에 x가 있으면 1을, 없으면 0을 출력한다. (1 ≤ x ≤ 20)

```
} else if (cmd == "check") {
    cin >> x;
    x--;
    int check = (s & (1 << x));
    cout << (check ? 1 : 0) << '\n';
```

11723

- `toggle x` : S에 x가 있으면 x를 제거하고, 없으면 x를 추가한다. (1 ≤ x ≤ 20)



11723

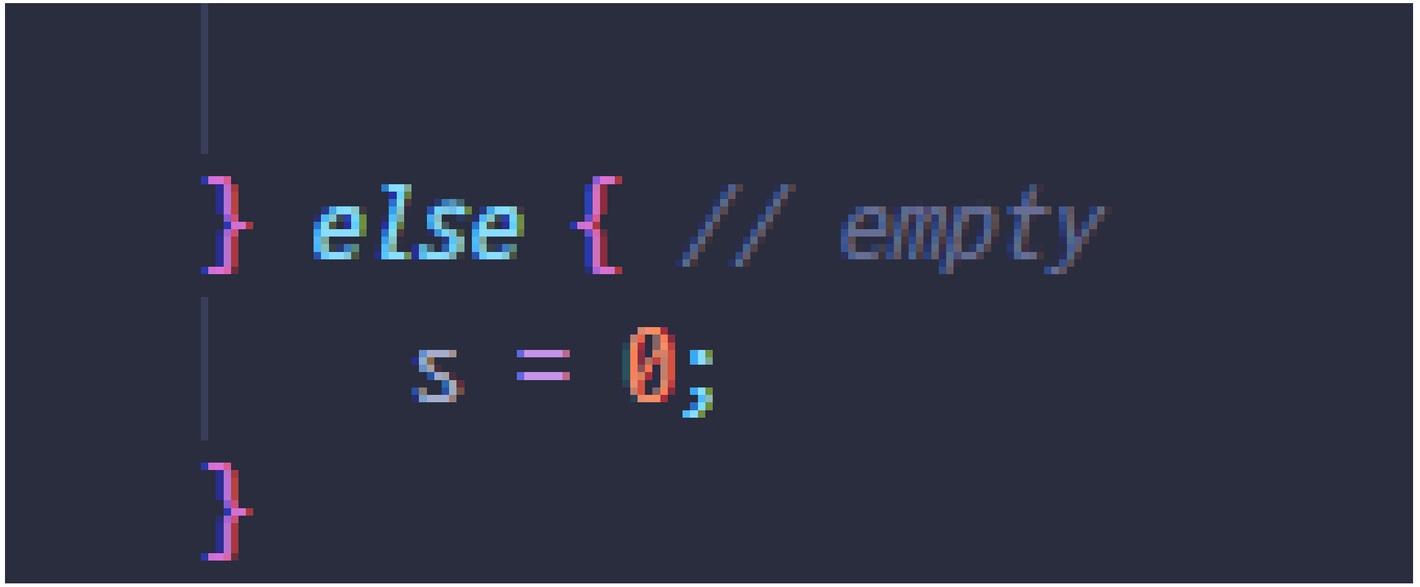
--

- all : S를 {1, 2, ..., 20} 으로 바꾼다.

```
} else if (cmd == "all") {
    s = (1 << N) - 1;
```

11723

- `empty` : S를 공집합으로 바꾼다.



문제

boj.kr/12813

boj.kr/25166

boj.kr/11723

boj.kr/15787

boj.kr/28239