비트마스킹
4주차

> 이진수

비트마스킹을 이해하기 위해서는 이진수를 이해해야 합니다.

우리가 평소에 표현하는 수는 0 ~ 9라는 수를 기반으로 수를 표현하는 "십진법"인데요.
먼저 십진법을 예로 들어보겠습니다.
123은 어떻게 이루어진 수일까요?
오른쪽 끝에서부터 보면 다음과 같습니다.
3 * 10 ^ 0
을 더하고
2 * 10 ^ 1
을 더하고
1 * 10 ^ 2
를 더한 것을 알 수 있습니다.
그리고 이 각각의 자리는 0 ~ 9로 10개의 숫자로 "표현"됩니다.

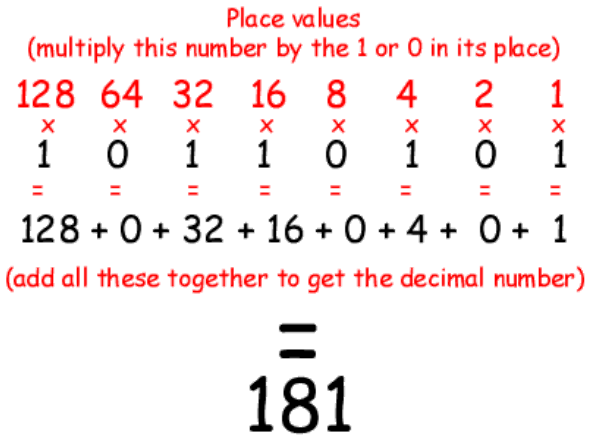
자 이제 그러면 이진수를 볼까요?
0 과 1, 두개의 숫자로 표현하는 "이진법"으로 표현하는 수가 바로 이진수라고 합니다.

일반적으로 이진법의 수를 십진법의 수와 구별하기 위해 다음과 같은 방법을 씁니다.
- 100101b (b를 덧붙임(b는 binary의 약자))(binary = 이진)
- $100101_{(2)}$ ((2)를 덧붙임, 주로 수학에서 쓰임)
- 0b100101 (앞에 0b를 덧붙임)

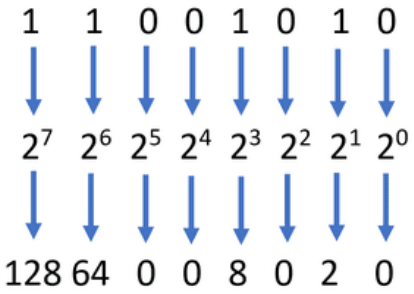
자 그러면 예제를 통해 10진수를 2진수로 표현해보며 이진수를 알아보겠습니다.

오른쪽 끝에서부터 각각의 자리는 1부터 2가 곱해지며 1, 2, 4, 8, 16, 32 ... 이런식으로 2배씩 증가하게 되며 수를 표현합니다.
각각의 자리는 "비트"라고 할 수 있으며 0인지 1인지를 통해 해당 수 1 또는 2, 4 등을 더하지 않거나 더하는 걸 기반으로 수를 표현합니다.

다음 그림은 181을 표현하는 이진수인 10110101 입니다.

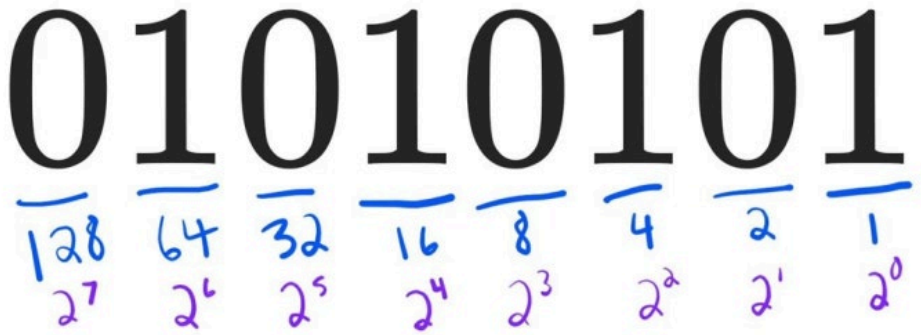


자, 그럼 다음수는 뭘뜻할까요? 11001010입니다. 각 자리수를 더하면 어떻게 될까요? 128 + 64 + 8 + 2가 되어 202라는 수가 나옵니다.



이렇게 각 비트가 나타내는 수는 오른쪽 끝부터 2의 0승, 2의 1승 2의 2승... 이 됩니다.

자, 다시 다음 그림은 어떤 수를 의미할까요? 64 + 16 + 4 + 1 = 85가 되겠죠?



자, 그러면 이 표가 이해가 가시나요?

[표2] 10진수의 2진수 표현

10진수	1	2	3	4	5	6	7	8	9
2진수	1	10	11	100	101	110	111	1000	1001

하나씩 써보면서 해보시면 이해가 가실 겁니다. 이렇게 이진수에 대해서 알아봤습니다.

> 이진수를 이용해 불리언배열을 표현할 수 있다.

예를 들어 4는 100, 2는 010, 1은 001 이죠?
1. 0번째 비트가 켜져있다 = 0번째 포함되어있다. = 001 = 1이 될 수 있죠?
2. 0, 1번째 비트가 켜져있다. = 0, 1번째 포함되어있다. = 011 = 3 이 될 수 있죠?

즉, 이진수를 이용해서 하나의 숫자를 기반, 불리언배열을 표현할 수 있습니다.

> 비트연산자 기초 +, -

비트연산자의 기초인 +와 -에 대해서 알아보겠습니다.

참고로 비트연산자라 함은 이거라서 +, -는 비트연산자로 볼 수는 없습니다. 다만 비트연산자를 활용할 때 -가 나오기 때문에 한번 배워봅시다.

&	비트단위로 AND 연산을 한다.
|	비트단위로 OR 연산을 한다.
^	비트단위로 XOR 연산을 한다.
~	피연산자의 모든 비트를 반전시킨다.
<<	피연산자의 비트 열을 왼쪽으로 이동시킨다.
>>	피연산자의 비트 열을 오른쪽으로 이동시킨다.

이는 십진법의 +와 -의 방식과 동일합니다.
5 + 1을 이진수로 표현해볼까요?
101 + 1이죠?
1이 더해지면서 윗자리로 1로써 옮겨지면서
110, 6이 됩니다.

이는 우리가 9 + 1 = 10이 되는 것과 동일합니다.
십진법은 10을 기수로 한 숫자표현법이고
이진법은 2를 기수로 한 숫자표현법이기 때문에 2를 기반으로 옮겨진다고 보시면 됩니다.

5 - 1을 이진수로 표현해볼까요?
101 - 1이죠?
그렇다면 100 = 4가 됩니다.

조금 더 어렵게 해볼까요?
10101 + 11(21 + 3)은 어떻게 될까요?

11000(24)이 됩니다.

10101 - 11(21 - 3)은 어떻게 될까요?
10010(18)이 되겠죠?

또한 아직 십진법 ~~를 이진법 ~~로 하는게 익숙하지 않다면 다음 사이트를 통해서 익히시면 됩니다.
이 사이트는 큰수를 넣지 않으면 16자리까지 표현되고 큰수를 넣으면 32자리까지 표현되는 것을 참고해주세요.

https://www.rapidtables.com/convert/number/decimal-to-binary.html

Decimal to Binary Converter
Decimal to Binary converter From To Enter decimal number 10 = ···
www.rapidtables.com

> 비트연산자 기초 &, |

자, 이번에는 &(AND)와 |(OR)입니다.
&는 true & true = true (1 & 1 = 1)고 나머지는 모두 false를 반환하는 것입니다.
1은 true고, false는 0이죠?

0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

만약 1001 & 1000을 하면 어떻게 될까요?
1000이 되겠죠? 모두 1이 아닌 부분에 대해서는 0이 되기 때문이죠.

|를 볼까요? 이는 하나라도 true면 모두 true가 됩니다.

0 | 0	0
0 | 1	1
1 | 0	1
1 | 1	1

만약 1001 | 1000을 하면 어떻게 될까요?
1001이 되겠죠? 1, 0은 1이 되니까요.
자, 그럼 또 문제.
1001과 0110을 하면 어떻게 될까요?
11111이 되겠죠?

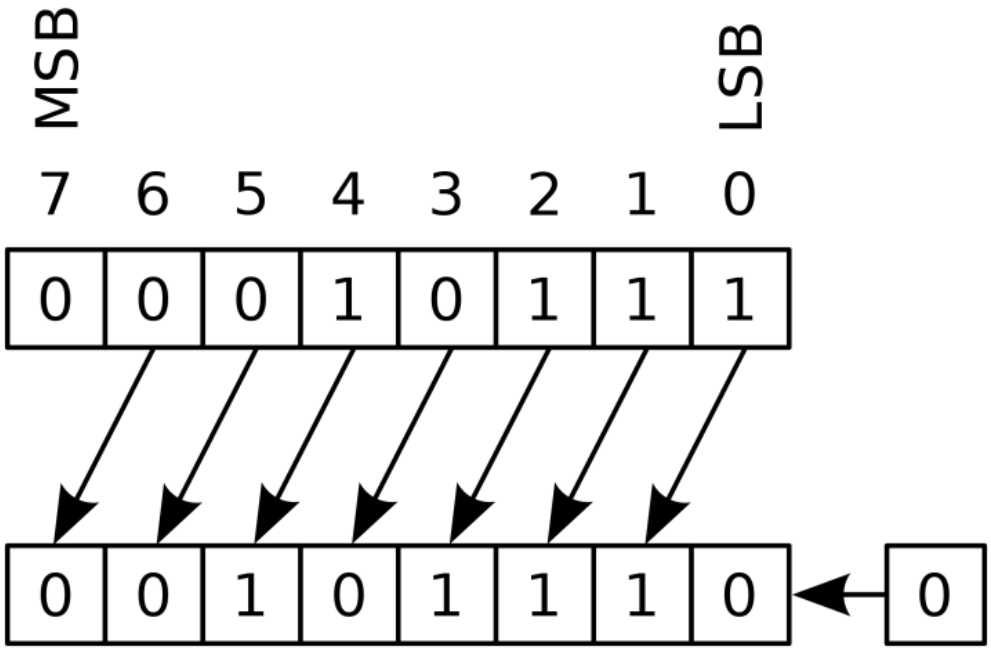
> 비트연산자 기초 〈〈, 〉〉

왼쪽 시프트 연산자 << , 오른쪽 시프트 연산자 >> 에 대해서 알아보겠습니다.
좀 더 자세히 설명하면, 사실 시프트 연산자는 logical shift, circlular shift 등이 있지만 C++ 에서는 <<라고 했을 때 logical shift로 되니 logical shift 만 배우셔도 됩니다.

따라서 정확한 워딩은 left logical shift, right logical shift이며
앞으로는 그냥 왼쪽 시프트 연산자, 오른쪽 시프트 연산자라고 할게요.

자 한번 배워보죠.

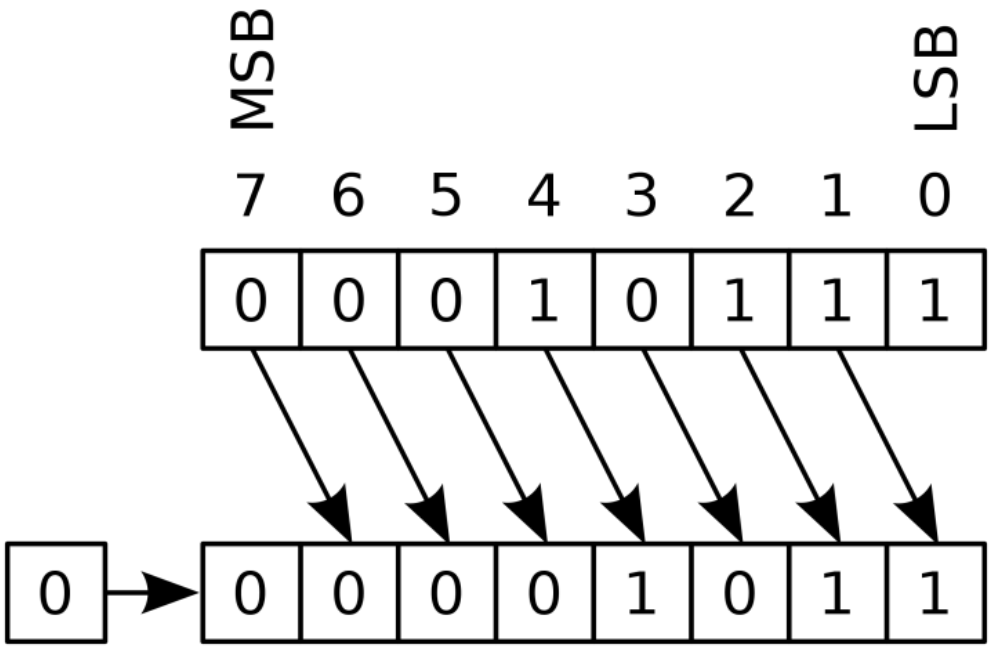
먼저 왼쪽 시프트 연산자 << 입니다.
비트를 왼쪽으로 옮기는 것을 볼 수 있죠? 다음 그림은 value << 1 을 한 모습입니다.
a << b는 a라는 비트를 b만큼 왼쪽으로 옮긴다는 의미입니다.



예를 들어 111(7) << 1 한다면 어떻게 될까요? 1110(14)이 되겠죠?
111(7) << 2를 한다면?
11100(28)이 됩니다.
즉, a << b 는 a * 2 ^ b와도 같은 의미를 가집니다.

비트마스킹할 때는 (1 << value)꼴로 많이 쓰이는데요.
1 << 3을 하면 뭘까요? 1000(8)이 되겠죠?
1 << 4를 하면? 10000(16)이 되는 것을 볼 수 있습니다.

오른쪽 시프트 연산자 >> 는 다음과 같이 비트를 오른쪽으로 옮기는 것을 말합니다. 다음그림은 오른쪽으로 1만큼 옮긴 모습입니다.
a >> b는 a라는 비트를 b만큼 오른쪽으로 옮긴다는 의미입니다.



101(5) >> 1을 하면 어떻게 될까요? 010(2)이 되겠죠?
1111(15) >> 2는요? 0011(3)이 됩니다.

a >> b 는 (int) a * (1 / 2) ^ b 와 같은 의미를 가집니다.
a >> b는 1 / 2... 하면서 외우시면 분수를 기반으로 외우시게 되서 헷갈릴 수도 있으니 비트를 오른쪽으로 빼는 걸 생각하며 외우시는 걸 추천드립니다.

> 비트연산자 기초 ^, ~

XOR연산자, ^입니다. true ^ true = false, false ^ false = false 이며 나머지는 다 true를 반환하는 녀석입니다.
같은 걸 싫어해~ 하면서 false(0)을 반환한다고 생각하시면 됩니다.

0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

자 그러면 제가 질문 드릴게용.
1001(9)^ 1000(8)은 어떻게 되죠?
0001이 되죠?
1010(10) ^ 1000(8)은요?
0010이 되죠?

~ 입니다. 1의 보수연산자이며 이는 해당 수의 모든 비트를 반전하는 연산자입니다.
~value = -(value + 1)이라는 특징을 갖습니다.

~ 0	1
~ 1	0

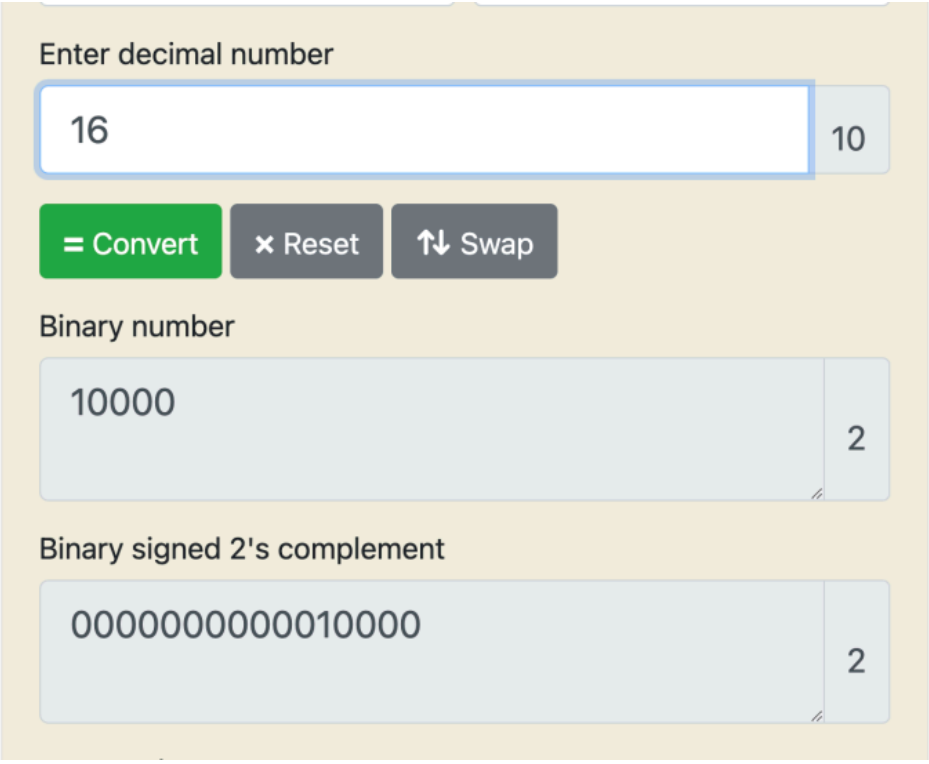
~011 하면 어떻게 될까요?
사실 011은
00000000,00000000,00000000,00000011이죠?
여기서 비트를 뒤집으면?
11111111 , 11111111 , 11111111 , 11111100이 됩니다.

이게 비트연산자를 통해서 음수가 나오니 음수에 대해 조금 설명해보겠습니다.
우리가 음수를 표현할 때는 2의 보수법을 이용합니다.
해당 양수의 모든 비트를 반전한 수에 1을 더하여 음수를 표현하는 방법인데요.

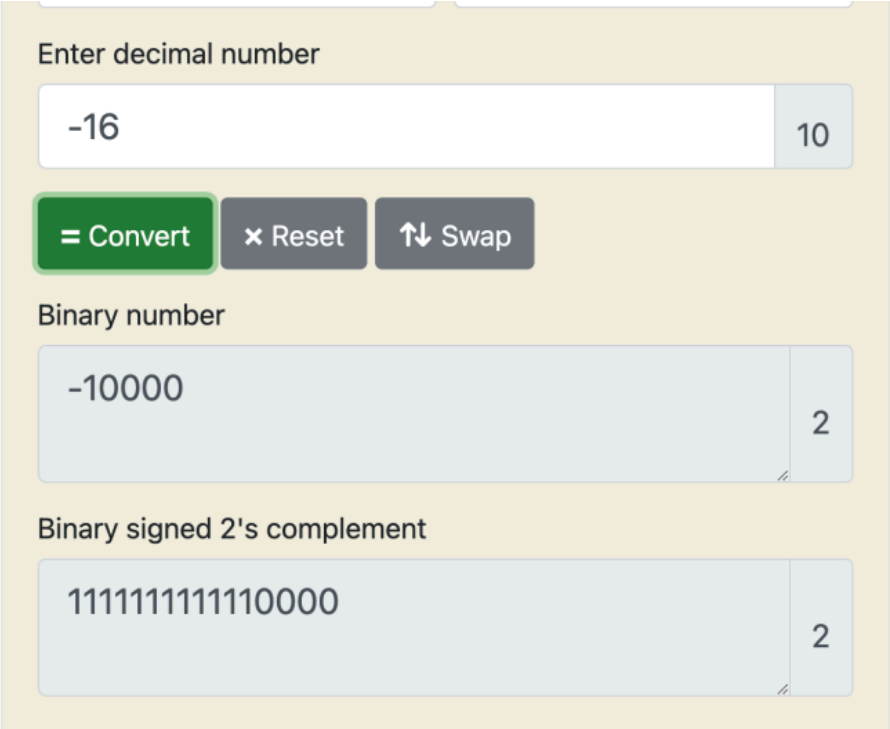
이 사이트는 큰수를 넣지 않으면 16자리까지 표현되고 큰수를 넣으면 32자리까지 표현되는 것을 참고해주세요.
https://www.rapidtables.com/convert/number/decimal-to-binary.html?x=16

16 decimal to binary conversion
Decimal to Binary converter From To Enter decimal number 10 = ···
www.rapidtables.com

이 사이트와 함께 설명해보겠습니다.
16은 뭐죠?
00000000,00000000,00000000,00010000 이죠?



여기서 ~를 통해 비트를 뒤집으면 뭐가되죠?
~00000000,00000000,00000000,00010000
=
11111111 , 11111111 , 11111111 , 11101111
이 될겁니다.자 여기서 +1을 해서 음수를 표현하는 것이 지금의 음수 표현법입니다.



즉, -value= ~value + 1이기 때문에
~ value = -(vallue + 1)이 됩니다.

> 비트연산자 활용법 #1. idx번째 비트끄기

사실 우리가 이제는 비트마스킹을 할 때 쓰이는 비트연산자 활용법에 대해서 알아보도록 할건데요.
아래의 이것들. 다 외워주셔야 합니다.
뭐 첨엔 어렵지만 하다보면 외워집니다. 하하..
천천히 알려드릴게요.

idx번째 비트끄기	S &= ~(1 << idx)
idx번째 비트 XOR 연산	S ^= (1 << idx)
최하위 켜져있는 비트 찾기	idx = (S & -S)
크기가 n인 집합의 모든 비트를 켜기	(1 << n) - 1
idx번째 비트를 켜기	S |= (1 << idx)
idx번째 비트가 켜져 있는지 확인하기	if(S & (1 << idx))

비트연산자 활용법 #1. idx번째 비트끄기 입니다...!

idx번째 비트끄기	S &= ~(1 << idx)

자, S = 10010(18)에서 저는 10010 이부분의 비트를 끄고 싶어요.
즉 1번째 비트죠?
참고로 1은 비트가 켜졌다. 0은 비트가 꺼진 겁니다.
어떻게 해야할까요?
먼저 000010을 만듭니다.
1 << 1을 하면 만들 수 있겠죠?
그 다음 비트를 뒤집어 줍니다.
111101 이런게 완성이 됩니다.
여기서
S = 10010 과 & 연산자를 하면 어떻게 될까요?
10010
&
111101을 하면?
10000이 되죠? 이렇게 해당 비트를 끌 수 있습니다.

코드

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int S = 18;
    int idx = 1;
    S &= ~(1 << idx);
    cout << S << '\n'; // 16
    return 0;
}
```

> 비트연산자 활용법 #2. idx번째 비트 XOR 연산(0은 1로, 1은 0으로)

---

idx번째 비트 XOR 연산	S ^= (1 << idx)

자, 이번에는 해당 idx번째의 비트에 XOR연산을 할겁니다.
그렇게 해서 무엇을 우리는 이룰것이냐??!!
해당 비트가 0이라면 1, 1이라면 0으로 만드는 걸 해볼껀데요.
S = 10010(18)에서
1번째 비트를 0으로 만들어볼게요.
(1 << 1)을 통해 00010을 만듭니다.
그리고
10010과
00010을 XOR연산자를 하면?
자 왼쪽에서부터

1 ^ 0 = 1
0 ^ 0 = 0
0 ^ 0 = 0
1 ^ 1 = 0
0 ^ 0 = 0
이렇게 되서
해당 부분만 바뀌어 10000이 되는 걸 볼 수 있죠?

자 그러면 이번에는 0을 1로 바꿔볼게요.
10010에서
0번째 비트는 0이죠?
(1 << 0)을 통해 00001을 만들어줍니다.
그리고 이를 ^ 연산자를 해줍니다.
1 ^ 0 = 1
0 ^ 0 = 0
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
짠,
10011이 되는 것을 볼 수 있죠?

코드

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int S = 18;
    int idx = 1;
    S ^= (1 << idx);
    cout << S << '\n'; // 16
    return 0;
}
```

> 비트연산자 활용법 #3. 최하위 켜져있는 비트 찾기

최하위 켜져있는 비트 찾기	idx = (S & -S)

자, 일단 최하위 켜져있는 비트가 뭘까요?
10010에서 최하위 켜져있는 비트는 1번째의 비트죠?
10000에서 최하위 켜져있는 비트는 4번째 비트죠?
오른쪽에서부터 탐색을 해서 1을 찾아서 해당 인덱스를 찾으면 됩니다.
이 인덱스 idx는
idx = (S & -S)를 통해 찾을 수 있습니다.
S = 10010이라고 할게요.
-S는 뭐죠?
~S = -(S + 1)이기 때문에

01101 + 1= -S
01110 = -S죠?
자 그럼 여기서
01110
&
10010
을 하면?
00010이 반환이 되죠?

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int S = 18;
    int idx = (S & -S);
    cout << idx << '\n'; // 2
    return 0;
}
```

> 비트연산자 활용법 #4. 크기가 n인 집합의 모든 비트를 켜기

크기가 n인 집합의 모든 비트를 켜기	(1 << n) - 1

자, 우리가 이진수부터 시작해서 비트연산자, 비트마스킹을 배우는 이유는 이러한 비트들을 하나의 배열처럼 쓰기 위함입니다.
예를 들어
1111은
0번째, 1번째, 2번째, 3번째 요소가 "포함"되어있다. 로 놓을 수 있습니다.
배열 크기가 4인 집합 [1, 1, 1, 1] 이 아니라 숫자 15 딱 하나만으로 이를 우리는 "표현"할 수 있죠.

자 그러면 n이 4, 즉 크기가 4인 배열의 모든 요소를 포함한다를 어떻게 나타내죠?
[1, 1, 1, 1]로 나타낼 수도 있겠죠?
뭐 여기서 1번째는 포함하지 않는다면..
[1, 1, 0, 1]이 될 거구요.
자 이를 비트연산자를 활용해서 표현해볼게요.
크기가 4라면
16 (1 << 4)가 되구요.
(1 << 4) - 1을 하게 되면
15가 나옵니다.
15는 앞서 설명한 1111이죠? 이렇게 크기가 4인 집합을 하나의 숫자로 나타낸다면 (1 << n) - 1을 하면 됩니다.
이진수로 계산을 해볼까요?
10000에서 1을 빼면
01111이 되니까요.

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n = 4;
    cout << (1 << n) - 1 << '\n'; // 15
    return 0;
}
```

> 비트연산자 활용법 #5. idx번째 비트를 켜기

---

idx번째 비트를 켜기	S |= (1 << idx)

10010(18)이 있습니다. 여기서 0번째 비트를 키면 어떻게 될까요?
10011(19)가 되죠?
이를
S |= (1 << idx)로 나타낼 수 있습니다.
1 << 0은
00001이며
이를
10010
|
00001을 하게 되면
왼쪽에서부터
1 | 0 = 1
0 | 0 = 0
0 | 0 = 0
1 | 0 = 1
0 | 1 = 1
이 되어서
10011이 됩니다. 쉽죠?
코드

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int S = 18;
    int idx = 0;
    S |= (1 << idx);
    cout << S << '\n'; // 19
    return 0;
}
```

비트연산자 활용법 #6. idx번째 비트가 켜져있는지 확인하기

idx번째 비트가 켜져있는지 확인하기	if(S & (1 << idx))

자, 10010에서 3번째 비트는 켜져있을까요?
아니죠.
이를
10010
&
00100을 만들어서 확인할 수 있습니다.
왼쪽에서부터
1 & 0 = 0
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
0 & 0 = 0
즉, 결과값은 00000이 되기 때문에 0, 즉 false를 반환하게 됩니다.

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int S = 18;
    int idx = 0;
    if(S & (1 << idx)){
        cout << "해당 idx : " << idx << "가 켜져있습니다.\n";
    }else{
        cout << "해당 idx : " << idx << "가 꺼져있습니다.\n";
    }
    return 0;
}
// 해당 idx : 0가 꺼져있습니다.
```

비트마스킹 : 경우의수, 매개변수

자 위의 것들을 외우고 이해했다면 이제 비트마스킹의 시간입니다.

여러분. 어떠한 특정원소를 찾을 때 어느정도의 시간복잡도는 어떻게 될까요? 바로 자료구조 등등 마다 다르겠죠?
- 배열의 어떤 요소를 찾을 때 선형적인 시간 : O(N)이 걸립니다.
- sorted array에서 이분탐색으로 찾을 때는 O(logN)이 걸립니다.
- 해싱테이블에서는 O(1)이 걸립니다.

자 그렇다면 불리언 배열에서 무언가를 찾는 등의 연산은 어떻게 될까요? 보통 불리언배열은 vector<bool>, bitset, set<int> 로 표현해서 처리하곤 합니다. 여기서 find메서드를 쓰는 등을 통해 연산을 하겠죠. 하지만

이러한 것들보다 비트연산을 쓰면 좀 더 가볍고 빠르겠죠?
이렇게 불리언배열의 역할을 하는 "하나의 숫자"를 만들어서 "비트 연산자"를 통해 탐색, 수정 등의 작업을 하는 것을 비트마스킹이라고 합니다.

자 앞서 설명한 것을 다시 설명해볼게요.
101은 1과 4의 합집합이며 이는 {0, 2}로 표현이 가능하죠?
111은? {0, 1, 2}로 표현이 가능하죠?

즉, 불리언배열을 만들어서 {0, 1, 0, 1} 이렇게 만들지 않고 **0101 이라는 하나의 수, 5** 등을 이용해 **하나의 숫자로 불리언 배열같은 역할**을 할 수 있습니다. 또한 앞서 배운 비트연산자를 통해 해당 요소가 포함되어있는지 안되어있는지 등을 쉽게 알 수 있죠.

참고 : 왜 비트마스킹이라고 하나요?

컴퓨터의 저장 공간인 메모리에는 0이나 1이 저장됩니다. 이러한 메모리의 용량을 표현하는 단위는 여러가지가 있는데 그 중 최소 단위를 비트(Bit)라고 하며 이 비트는 0 또는 1이라는 값을 가집니다. 이를 기반으로 해당 비트를 켜거나(1로 만들거나) 끄거나(0으로 만들거나) 하는 "마스킹"하기 때문에 비트마스킹이라고 합니다.
참고로 비트는 이진수의 약어입니다. 비트나 이진수나 똑같은 개념이라고 보시면 됩니다. (BIT = Binary Digit)

> 비스마스킹을 이용한 경우의 수

비스마스킹은 경우의 수를 표현하는데도 잘 쓰입니다.
예를 들어 {사과, 딸기, 포도, 배}의 모든 경우의 수는 어떻게 될까요?
{사과}
{딸기, 사과}
..
이렇게 총 16가지수가 되겠죠? 사과를 포함하거나 포함하지 않거나 해서 각각의 요소는 2개의 상태값을 가지기 때문에 2^4 = 16이 될 겁니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int n = 4;
int main() {
    string a[n] = {"사과", "딸기", "포도", "배"};
    for(int i = 0; i < (1 << n); i++){
        string ret = "";
        for(int j = 0; j < n; j++){
            if(i & (1 << j)){
                ret += (a[j] + " ");
            }
        }
        cout << ret << '\n';
    }
    return 0;
}
/*

사과
딸기
사과 딸기
포도
사과 포도
딸기 포도
사과 딸기 포도
배
사과 배
딸기 배
사과 딸기 배
포도 배
사과 포도 배
딸기 포도 배
사과 딸기 포도 배
*/
```

이렇게 모든 집합을 표현한 것을 볼 수 있습니다.
i는 0000, 0001, 0010을 상징합니다.
j는 0, 1, 2, 3을 기반으로 (1 << 0), (1 << 1) 등으로 해당 번째의 비트가 켜져있냐 켜져있지 않냐를 통해 집합을 확인합니다.
즉 이러한 것을 통해 4C1, 4C2 ... 이러한 조합들의 모든 경우의 수를 한번의 for문 만으로 표현이 가능한 셈이죠.
combi라는 함수를 통해 조합을 할 수 있다고 설명했는데요.
문제에서 4C1, 4C2 여러가지 조합 등 모든 경우의 수를 기반으로 로직을 짜야 하는 경우 유용한 셈이죠.

비트마스킹을 이용한 매개변수 전달

사과라는 매개변수가 포함이 되어있고 이어서 사과 + 포도, 사과 + 배 이런식의 매개변수를 더하는 것을 구현하고 싶다면 이렇게 하면 됩니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
const int n = 4;
string a[4] = {"사과", "딸기", "포도", "배"};
void go(int num){
    string ret = "";
    for(int i = 0; i < 4; i++){
        if(num & (1 << i)) ret += a[i] + " ";
    }
    cout << ret << '\n';
    return;
}
int main() {
    for(int i = 1; i < n; i++){
        go(1 | (1 << i));
    }
    return 0;
}
/*
사과 딸기
사과 포도
사과 배
*/
```

마치며

이처럼 << 또는 & 등 비트연산자 등으로 불리언배열의 역할을 하는 비트마스킹을 알아보았습니다.
하지만 비트마스킹은 한계가 있습니다. 바로 31까지 가능합니다. int형 숫자의 한계까지인 셈이지요. long long은 어떠냐 하는 의견도 있는데 보통 2^ 30승정도만 해도... 10억이 되기 때문에 그 이후의 경우의 수를 센다는 것 자체가 이미 시간복잡도를 많이 초과하기 때문에 보통은 30 ~ 31 까지의 경우의 수만을 표현할 수 있다고 볼 수 있습니다.

https://blog.naver.com/jhc9639/222602625841

코딩테스트 강의 추천 10주완성 C++코딩테스트

안녕하세요. AI서비스를 만들고 있는 예비창업가이자 3년차 알고리즘 강…

blog.naver.com