해밍 코드 (Hamming Code)

컴퓨터 내부에서 데이터가 한 저장장치에서 다른 장소로 이동하거나, 혹은 통신 시스템을 통하여 컴퓨터들 간에 데이터가 전송될 때, 데이터가 손상(데이터 비트가 1에서 0으로, 혹은 0에서 1로 변경되는 경우 등)되는 경우가 발생한다. 이러한 데이터의 손상을 검출하는 가장 간단한 방법은 패리티 비트를 데이터에 추가하여 전송하는 방법이다(문제 "패리티 비트" 참조). 그러나 패리티 비트는 한 개의 비트에서 오류가 발생하면 그 오류를 검출할 수 있을 뿐이며, 두 개 이상 비트에서 오류가 발생하면 그 오류를 검출하지 못한다. 또한 전송된 데이터의 한 개의 비트에서 오류가 발생하였다는 것을 검출하더라도 어느 비트에서 오류가 발생하였는지는 판단하지 못한다. 이러한 패리티 비트의 단점을 보완하는 방법 중에는 해밍(Richard W. Hamming)이 개발한 해밍 코드(Hamming code)가 있다. 해밍 코드를 이용하면 하나의 비트에서 오류가 발생하는 경우에는 그 비트의 위치를 정확하게 알 수 있으며, 또한 두 개의 비트에서 오류가 발생하는 경우에도 그 오류를 검출할 수 있다.

다음 그림에서와 같이 해밍 코드에는 데이터 내부에 여러 개의 패리티 비트가 첨가되어 있다. 다음은 일반적인 해밍 코드에서 패리 비트를 사용하는 방법에 대하여 설명한다. 데이터에서 가장 낮은 자리의 비트의 위치를 1로 정하고, 자리가 한 자리씩 높아질수록 그 위치를 2, 3, 4, … 으로 부여한다.

1. 데이터에서 2의 거듭제곱이 되는 위치(예를 들어, 1, 2, 4, 8, 16, 32, 64 등)의 비트는 모두 패리티 비트로 사용된다.
2. 다른 모든 위치의 비트는 실제 데이터를 위하여 사용된다.
3. 각 위치의 패리티 비트는 해밍 코드의 특정 위치에 있는 비트들의 패리티를 나타낸다. 위치가 1, 2, 4, 8, 16 등에 있는 패리티 비트를 P_1, P_2, P_4, P_8, P_{16} 등으로 나타낼 때, 이 패리티 비트가 검사하는 비트의 위치는 다음과 같다(아래 그림 참조).
   3.1. P_1 : 위치가 1인 비트에서 시작하여, 한 비트를 검사하고, 한 비트를 건너 뛰고, 한 비트를 검사하고, 한 비트를 건너 뛰어 검사하는 것을 반복한다. 단, 위치가 1인 비트(P_1 자신)는 검사하지 않는다.
   3.2. P_2 : 위치가 2인 비트에서 시작하여, 두 비트를 검사하고, 두 비트를 건너 뛰고, 두 비트를 검사하고, 두 비트를 건너 뛰어 검사하는 것을 반복한다. 단, 위치가 2인 비트(P_2 자신)는 검사하지 않는다.
   3.3. P_4 : 위치가 4인 비트에서 시작하여, 네 비트를 검사하고, 네 비트를 건너 뛰고, 네 비트를 검사하고, 네 비트를 건너 뛰어 검사하는 것을 반복한다. 단, 위치가 4인 비트(P_4 자신)는 검사하지 않는다.
   3.4. P_8 : 위치가 8인 비트에서 시작하여, 여덟 비트를 검사하고, 여덟 비트를 건너 뛰고, 여덟 비트를 검사하고, 여덟 비트를 건너 뛰어 검사하는 것을 반복한다. 단, 위치가 8인 비트(P_8 자신)는 검사하지 않는다.
   3.5. P_{16} : 위치가 16인 비트에서 시작하여, 열 여섯 비트를 검사하고, 열 여섯 비트를 건

너 뛰고, 열 여섯 비트를 검사하고, 열 여섯 비트를 건너 뛰어 검사하는 것을 반복한다. 단, 위치가 16인 비트(P_{16} 자신)는 검사하지 않는다.

3.6. 다른 위치에 있는 패리티 비트들도 이와 같은 방법으로 패리티를 계산한다.

아래 그림은 32-비트 해밍 코드에서 각 패리티 비트의 위치와 각 패리티 비트 P_1, P_2, P_4, P_8, P_{16} 이 검사하는 비트의 위치를 • 으로 표시한 것이다. 이 그림의 예에서 보듯이 각 패리티 비트가 검사하는 비트에는 자신을 포함한 어떠한 다른 패리티 비트도 포함되어 있지 않으며, 패리티 비트를 제외한 실제 데이터에 속하는 비트만을 검사하는 것을 알 수 있다. 32-비트 해밍 코드에서는 아래 그림에서와 같이 패리티 비트를 제외한 실제 데이터는 최대 26-비트만을 사용할 수 있다.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	비트위치
P																P								P				P		P	P	패리티비트 위치 및 데이터 위치
	•		•		•		•		•		•		•		•		•		•		•		•		•		•		•		P	P_1 에서 검사하는 비트
	•	•			•	•			•	•			•	•			•	•			•	•			•	•			•		P		P_2 에서 검사하는 비트
	•	•	•	•					•	•	•	•					•	•	•	•					•	•	•	P					P_4 에서 검사하는 비트
	•	•	•	•	•	•	•	•									•	•	•	•	•	•	•	P									P_8 에서 검사하는 비트
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	P																		P_{16} 에서 검사하는 비트

예를 들어, 전송하고자 하는 실제 데이터가 다음과 같은 경우에 있어서

$$47367638_{(10)} = 10\ 1101\ 0010\ 1100\ 0101\ 1101\ 0110_{(2)}$$

짝수 패리티 기법을 사용할 경우 각 패리티 비트를 계산하면 다음 그림과 같다. 짝수 패리티 기법은 패리티 비트를 포함하여 검사하고자 하는 비트에서 비트 1의 개수가 짝수 개가 되도록 패리티 비트를 정하는 경우를 말한다. 따라서, 패리티 비트 P1이 검사하는 비트에서 비트 1의 개수는 9개이므로 P_1은 1이 되어야 한다. 같은 방법으로 계산하면, P_2, P_4, P_8, P_{16} 이 검사하는 비트에서 비트 1의 개수는 각각 6, 11, 9, 7개 이므로, 각 패리티 비트는 각각 0, 1, 1, 1이 되어야 한다.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	비트위치
P	1	0	1	1	0	1	0	0	1	0	1	1	0	0	0	P	1	0	1	1	1	0	1	P	0	1	1	P	0	P	P	데이터 및 패리티비트 위치
	1		1		0		0		1		1		0		0		1		1		1		1		0		1		0		1	P_1 에서 검사하는 비트
	1	0			0	1			1	0			0	0			1	0			1	0			0	1			0	0		P_2 에서 검사하는 비트
	1	0	1	1					1	0	1	1					1	0	1	1					0	1	1	1				P_4 에서 검사하는 비트
	1	0	1	1	0	1	0	0									1	0	1	1	1	0	1	1								P_8 에서 검사하는 비트
	1	0	1	1	0	1	0	0	1	0	1	1	0	0	0	1																P_{16} 에서 검사하는 비트

그러므로, 패리티 비트를 고려한 실제 데이터를 해밍 코드로 변환하면

$$101\ 1010\ 0101\ 1000\ \underline{11}01\ 1101\ \underline{1}011\ \underline{1}001_{(2)} = 1515773369_{(10)}$$

이 된다. 위에서 밑줄 친 비트가 패리티 비트이며, 그 이외의 비트는 실제 데이터 비트이다.

32-비트 해밍 코드로 만들어진 데이터를 수신하는 수신 측에서 수신된 데이터에 데이터의 손상이 있는 지를 검사하는 방법은 다음과 같다.

1. 모든 패리티 비트 P_1, P_2, P_4, P_8, P_{16} 에 대하여, 각 패리티 비트가 검사하여야 하는 비트의 패리티가 정확한지를 검사한다.
2. 만약, 패리티 비트 P_1, P_2, P_4, P_8, P_{16} 중에서 한 개의 패리티 비트에서만 패리티 오류가 발생하면, 그 패리티 비트 자체에 손상이 있는 경우이다. 즉, 실제 데이터에는 손상이 일어나지 않은 경우이다.
3. 그렇지 않고, 한 개 이상의 패리티 비트에서 패리티 오류가 발생하면, 오류가 난 패리티 비트를 이용하여 손상된 비트의 위치를 계산할 수 있다. 예를 들어, 아래 그림에서와 같이 26번 비트가 손상이 되면, 26번 비트를 검사하여야 하는 모든 패리티 비트 P_2, P_8, P_{16} 에서 패리티 오류가 발생한다.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	비트위치
P	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0	P	1	0	1	1	1	0	1	P	0	1	1	P	0	P	P	데이터 및 패리티비트 위치
	1		1		0		0		1		1		0		0		1		1		1		1		0		1		0		1	P_1
	1	0			0	0			1	0			0	0			1	0			1	0			0	1			0	0		P_2 오류
	1	0	1	1					1	0	1	1					1	0	1	1					0	1	1	1				P_4
	1	0	1	1	0	0	0	0									1	0	1	1	1	0	1	1								P_8 오류
	1	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1																P_{16} 오류

4. 오류가 발생한 패리티 비트 P_2, P_8, P_{16}을 이용하면, 다음과 같이 손상된 비트의 위치를 알 수 있다. 즉, 패리티 오류가 발생한 패리티 비트의 위치인 2, 8, 16을 모두 더한 수 2+8+16=26이 오류가 발생한 비트의 위치가 된다.

	P_{16}	P_8	P_4	P_2	P_1	
이진수	1	1	0	1	0	
십진수	16	8		2		Σ = 26

5. 손상된 위치 26번의 비트를 0에서 1로 바꾸고, 패리티 비트를 모두 제거하면, 원래의 정확한 데이터 10 1101 0010 1100 0101 1101 $0110_{(2)}$을 구할 수 있다.
6. 다음으로는 두 개의 데이터 비트가 손상이 된 경우를 고려해보자. 예를 들어, 7번째 비트와 31번째 비트가 손상이 된 경우에는, 5번째 비트는 P_1, P_2, P_4 에 의하여 검사되고, 29번째 비트는 P_1, P_4, P_8, P_{16} 에 의하여 검사된다. 따라서, 7번째 비트와 29번째 비트를 동시에 검사하는 P_1, P_4 의 패리티 비트에는 오류가 발생하지 않고, P_2, P_8, P_{16} 에 패리티 오류가 발생한다. 그러나, 이 경우에서는 26번 한 개의 비트에 오류가 발생한 것과 동일 한 패리티 오류가 발생하므로, 오류가 발생한 것은 검출할 수 있으나, 한 비트에 오류가 발생한 것인지 두 개의 비트에서 오류가 발생한 것인 지를 구분할 수 없다.

32-비트 해밍 코드를 이용하여 데이터를 전송하는 경우에, 주어진 데이터를 32-비트 해밍 코드로 변환하거나, 32-비트 해밍 코드에서 실제 데이터를 추출하는 프로그램을 작성하시오. 단, 해밍 코드에서 오류가 있는 경우에는, 하나의 비트에 손상이 된 경우를 가정하여, 손상된 해밍 코드를 복구하여 실제 데이터를 추출한다. 또한 해밍 코드의 패리티 비트는 짝수 패리티 기법을 사용하여 그 값을 결정한다고 가정한다.

입력

입력은 표준입력(standard input)을 사용한다. 입력은 t 개의 테스트 케이스로 주어진다. 입력 파일의 첫 번째 줄에 테스트 케이스의 개수를 나타내는 정수 t 가 주어진다. 두 번째 줄부터 t 개의 줄에는 한 줄에 한 개의 테스트 케이스에 해당하는 두 개의 정수 k n ($k \in \{0, 1\}$) 이 주어진다. 첫 번째 정수 k 가 0인 경우는 두 번째 데이터 n ($0 \leq n \leq 2^{26}-1$) 이 전송하여야 할 실제 데이터인 경우이고, 첫 번째 정수 k 가 1인 경우는 두 번째 데이터 n ($0 \leq n \leq 2^{31}-1$) 이 실제 데이터를 추출하여야 할 32-비트 해밍 코드인 경우이다. 두 정수 사이에는 한 개의 공백이 있으며, 잘못된 데이터가 입력되는 경우는 없다.

출력

출력은 표준출력(standard output)을 사용한다. 입력되는 테스트 케이스의 순서대로 다음 줄에 이어서 각 테스트 케이스의 결과를 출력한다. 각 테스트 케이스에 해당하는 첫 번째 정수 k 가 0인 경우는 두 번째 데이터 n 으로 부터 만들어진 32-비트 해밍 코드를 출력하고, 첫 번째 정수 k 가 1인 경우에는 두 번째 데이터인 32-비트 해밍 코드 n 으로 부터 26-비트의 실제 데이터를 추출한다. 해밍 코드에 오류가 있는 경우에는, 하나의 비트에 손상이 있는 경우를 가정하여, 손상된 해밍 코드를 복구하여 추출한 실제 데이터를 출력한다.

입력과 출력의 예

입력	출력
3 0 47367638 1 1515773369 1 1482218937	1515773369 47367638 47367638

MyHammingCode.h

```
#ifndef _MY_HAMMING_CODE_H_
#define _MY_HAMMING_CODE_H_

class MyHammingCode
{
public:
    // constructors
    MyHammingCode ();
    MyHammingCode (unsigned int val);

    // accessor/mutator functions
    unsigned int getEncodedValue();
    unsigned int getDecodedValue();

private:
    unsigned int value;
    unsigned int encodedValue;
    unsigned int decodedValue;

    unsigned int copyBits(unsigned int n, int from, int num, int to);
    unsigned int setParityBit(unsigned int n, int pos);
    unsigned int checkParityBit(unsigned int n);
    int hammingWeight(unsigned int n);
    unsigned int setBit(unsigned int n, int pos);
    unsigned int clearBit(unsigned int n, int pos);
    int checkBit(unsigned int n, int pos);
    unsigned int negateBit(unsigned int n, int pos);
    unsigned int _xor(unsigned int x, unsigned int y);
};

#endif // _MY_HAMMING_CODE_H_
```

MyHammingCode.cpp

```
#include "MyHammingCode.h"

// constructors
MyHammingCode::MyHammingCode ()
:value(0)
{
}

MyHammingCode::MyHammingCode (unsigned int val)
:value(val)
{
}

// utility functions
unsigned int MyHammingCode::getEncodedValue ()
{
    encodedValue = 0;

    encodedValue |= copyBits(value, 0,  1,  2);
    encodedValue |= copyBits(value, 1,  3,  4);
    encodedValue |= copyBits(value, 4,  7,  8);
    encodedValue |= copyBits(value, 11, 15, 16);

    encodedValue = setParityBit(encodedValue, 1);
    encodedValue = setParityBit(encodedValue, 2);
    encodedValue = setParityBit(encodedValue, 4);
    encodedValue = setParityBit(encodedValue, 8);
    encodedValue = setParityBit(encodedValue, 16);

    return encodedValue;
```

```
}

unsigned int MyHammingCode::getDecodedValue ()
{
    int errorPos;

    decodedValue = 0;

    errorPos = checkParityBit(value);
    if (errorPos > 0)
        value = negateBit(value, errorPos-1);

    decodedValue |= copyBits(value, 2, 1, 0);
    decodedValue |= copyBits(value, 4, 3, 1);
    decodedValue |= copyBits(value, 8, 7, 4);
    decodedValue |= copyBits(value, 16, 15, 11);

    return decodedValue;
}

/*
 * 정수 n의 from-번째 비트부터 num개의 비트를
 * 다른 정수 num의 to-번째 비트로 복사하는 함수
 */
unsigned int MyHammingCode::copyBits(unsigned int n, int from, int num, int to)
{
    unsigned int mask = 0;
    unsigned int code = 0;

    /* num 비트 만큼의 mask 를 만듬 */
    do
    {
        mask <<= 1;
        mask |= 0x01;
    } while (--num);


    /* n의 from-번째 비트부터 num개의 비트를 골라내어 복사함 */
    n >>= from;
    n &= mask;
    code |= n;
    code <<= to;

    return code;
}

/* 패리티 비트를 만드는 함수 */
unsigned int MyHammingCode::setParityBit(unsigned int n, int pos)
{
    unsigned int mask[5] = { 0x55555554, 0x66666664, 0x74747470,
                             0x7F807F00, 0x7FFF0000 };

    int count;
    unsigned int checkBits;

    switch (pos)
    {
    case 1:
        checkBits = n & mask[0];
        break;
    case 2:
        checkBits = n & mask[1];
        break;
    case 4:
        checkBits = n & mask[2];
```

```
        break;
    case 8:
        checkBits = n & mask[3];
        break;
    case 16:
        checkBits = n & mask[4];
        break;
    }

    count = hammingWeight(checkBits);
    if (count % 2)
        n = setBit(n, pos-1);

    return n;
}

/*
 * 패리티 비트를 검사하는 함수
 * 단, 에러가 발생하는 경우에는 한 비트에서만 오류가 발생한 것을 가정하여,
 * 에러가 발생한 비트의 위치를 찾아서 리턴해준다.
 */
unsigned int MyHammingCode::checkParityBit(unsigned int n)
{
}

/* 정수에서 비트가 1인 자리수의 개수를 계산하는 함수 */
int MyHammingCode::hammingWeight(unsigned int n)
{
}

/* 정수의 pos 자리수의 비트를 1로 만드는 함수 */
unsigned int MyHammingCode::setBit(unsigned int n, int pos)
{
}

/* 정수의 pos 자리수의 비트를 0으로 만드는 함수 */
unsigned int MyHammingCode::clearBit(unsigned int n, int pos)
{
}

/* 정수의 pos 자리수가 0인지 1인지를 확인하는 함수 */
int MyHammingCode::checkBit(unsigned int n, int pos)
{
}


/* 정수의 pos 자리수를 반대로 만드는 함수 */
unsigned int MyHammingCode::negateBit(unsigned int n, int pos)
{
}

/* Exclusive OR 를 계산하는 함수 */
unsigned int MyHammingCode::_xor (unsigned int x, unsigned int y)
{
    return !x ^ !y;
}
```

TestMyHammingCode.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "MyHammingCode.h"
using namespace std;
```

```
int main()
{
    int numTestCases;

    cin >> numTestCases;

    for(int i=0; i<numTestCases; i++)
    {
        int type;
        unsigned int num;

        cin >> type >> num;

        MyHammingCode hamCode(num);

        if (type == 0)
            cout << hamCode.getEncodedValue() << endl;
        else
            cout << hamCode.getDecodedValue() << endl;
    }

    return 0;
}
```