**Como trabajar con máscaras de bits**

**Alejandro Hernández Gómez**

En programación, a parte de los operadores clásicos que trabajan con números (suma, resta, multiplicación, etc) existen otros operadores que trabajan con la representación en binario de los números. Estos operadores, llamados en inglés, *bitwise operators*, lo que hacen es operar sobre los números bit a bit. El primer bit del resultado será el resultado de operar el primer bit de cada entrada, el segundo bit será el resultado de operar los segundos bits de cada entrada, etc etc.  
  
Ejemplo: 134 & 42 = 2:

...010000110 ...000101010 ------------ AND ...000000010

(los ... indican que el número en realidad son 32 bits (o 64 si son long long).  
  
A continuación está la lista de todos los operadores de bits:  
  
**AND - &**  
Aplica la operación AND a cada bit. Es decir, el resultado es 1 si los dos bits de entrada son 1.  
  
**OR - |**  
Aplica la operación OR a cada bit. Es decir, el resultado es 1 si alguno de los bits de entrada es 1 (o los 2).  
  
**XOR - ^**  
Aplica la operación XOR a cada bit. Es decir, el resultado es 1 si exactamente uno de los bits de entrada es 1.  
  
**NOT - ~**  
Este operador invierte todos los bits. Es decir, intercambia 0 y 1. A diferencia de todos los otros operadores, este sólo usa 1 argumento. Se escribe antes del número. Por ejemplo: "~42".   
  
**Bit shifting: << y >>**  
Estos operadores lo que hacen es "desplazar" los bits de un número a la izquierda o a la derecha. Los bits que se "salen" del número se pierden, y rellena los bits que quedan con 0. (Excepto el operador right shift (>>) con números con signo, que lo que hace es "extender" el bit más alto para conservar el signo, pero esto normalmente no importa)  
  
Es equivalente a multiplicar o dividir por una potencia de 2. (Nota: un bit shift es más eficiente que una multiplicación o división, pero usarlo no hará nuestro código más eficiente. El compilador es inteligente y sustituye automáticamente todas las multiplicaciones o divisiones por potencias de 2 por bit shifts!)  
  
Ejemplo: 3 << 2 = 12

...000011 --------- << 2 ...001100

El uso más inmediato de estos operadores es para manipular bits directamente de un número. Podemos consultar el valor de un bit directamente, ponerlo a 0, ponerlo a 1, etc etc.  
  
**Consultar el valor de 1 bit**  
  
Primero, creamos un número que sea todo 0, y un 1 en el bit que queremos mirar. Es muy fácil: "1<<n".  
Luego, hacemos AND con el número que queremos mirar el bit. Si ese bit es 0, el resultado sera 0. Si no, no.  
  
El número "1<<n" actua como una máscara, es decir, "deja pasar" sólo el bit que queremos mirar.  
  
Por lo tanto:

**if**(x & (1<<n)) cout<<*"El bit es 1"*;  
**else** cout<<*"El bit es 0"*;

**Poner un bit a 1**  
También es fácil: basta con hacer un or del número con 1<<n.

x |= 1<<n;

**Poner un bit a 0**  
Este es el caso más complicado, pero aun así es fácil. Utilizaremos una máscara que tenga todo 1, y un 0 en el bit que queremos quitar, y la aplicamos con el AND.   
  
Para construir la máscara, hacemos 1<<n, y invertimos el resultado con el NOT.

x &= ~(1<<n);

**Usos de las máscaras de bits**  
Las máscaras de bits son muy útiles porque nos permiten hacer directamente operaciones que, de otra manera, necesitaríamos hacer un for y recorrer vectores.   
  
Por ejemplo, un número entero sirve para sustituir un vector<bool> de hasta tamaño 32 (o 64 con long long). O un set<int> con números hasta 32 o 64.  
  
Si representamos dos conjuntos con enteros A y B:  
-> A & B nos da la intersección de los conjuntos.  
-> A | B nos da la unión de los conjuntos.  
-> A & ~B elimima todos los elementos de B del conjunto A.  
etc etc.  
  
También sirven para utilizarlas como índices en vectores. De esta manera, podemos sustituir un map<set<int>, algo> por un vector<algo> utilizando números que representan los conjuntos como índices. Como es lógico, será \*mucho\* más eficiente. También facilitan la tarea en problemas de programación dinámica donde el "estado" es un conjunto.  
  
Hay que tener en cuenta que las máscaras de bits sirven en casos donde podemos guardar la información como enteros y trabajar con ellos. Construirlos es poco eficiente, ya que tenemos que hacer un for. No sirve de nada, para calcular la intersección de dos conjuntos, convertirlos a enteros, hacer un AND, y convertirlos a vectores otra vez, incluso será más lento que hacerlo directamente. Sí que sirve convertirlo todo a enteros al principio del algoritmo, hacer muchos cálculos utilizando los operadores con bits, y luego escribir el resultado.  
  
**Ejemplo 1**  
  
También pueden servir para programar ciertos problemas de manera mas rápida/fácil.  
Por ejemplo, el problema [problem=ceros-y-unos].  
Este problema está pensado para que se haga con recursividad.  
Pero, si nos fijamos, lo único que están pidiendo son los números desde 0 hasta 2n, en binario. Viendo esto, lo podemos hacer con un for, escribiendo cada número en binario, utilizando el método arriba explicado para conocer el valor de un bit concreto.

#include<iostream>  
using namespace std;  
  
**int** main()  
{  
        **int** n;  
        cin>>n;  
          
        *//Recorremos todos los numeros desde 0 hasta 2^n-1*  
        **for**(**int** i = 0; i < 1<<n; i++)  
        {  
                *//Escribimos i en binario, con n dígitos.*  
                **for**(**int** j = n-1; j >= 0; j--)  
                {  
                        **if**(i & (1<<j)) cout<<*"1"*;  
                        **else** cout<<*"0"*;  
  
                        **if**(j != 0) cout<<*" "*;  
                }  
  
                cout<<endl;  
        }  
}

Fácil, no? Nos hemos ahorrado una función recursiva, y además, esta manera de hacerlo seguramente sería un poco más eficiente.  
  
**Ejemplo 2**  
  
Problema [problem=minecraft-waterfalls]. Este problema se puede hacer con máscaras de bits bastante bien.   
La idea es guardar dos números por cada fila. Uno contiene qué posiciones tienen agua, y qué posiciones tienem bloques. Se guarda la última fila, se lee la nueva fila, y con unas pocas operaciones, puedes calcular qué casillas tienen agua, y puedes ir escribiendo la salida a medida que la lees.

#include<iostream>  
#include<vector>  
#include<fstream>  
  
using namespace std;  
  
*//A indica donde hay agua.*  
*//B indica donde hay bloques.*  
*//Devuelve true si se ha podido leer la linea.*  
bool leerLinea(**int** n, **long** long& a, **long** long& b)  
{  
        a = 0;  
        b = 0;  
          
        **char** x;  
          
        **for** (**int** i = 0; i < n; i++){  
                **if**(!(cin >> x)) **return** **false**;  
  
                **if** (x == *'F'*)  
                {  
                        a |= 3LL<<i;  
                        **if**(a != 0) *//No se puede hacer un shift negativo.*  
                                a |= 1LL<<(i-1);  
                }  
                **if** (x == *'#'*)  
                        b |= 1LL<<i;  
        }  
  
        *//Quitar agua donde haya bloques.*  
        a &= ~b;  
          
        **return** **true**;  
}  
  
**int** main(){  
        cout.sync\_with\_stdio(**false**);  
        **int** n;  
        cin >> n;  
        *//a1, a2 indican donde hay agua en la fila anterior y esta.*  
        *//b1, b2 indican donde hay bloques.*  
        **long** **long** a1 = 0, a2 = 0, b1 = 0, b2 = 0;  
  
        *//Leer la primera linea*  
        leerLinea(n, a1, b1);  
  
        **while** (leerLinea(n, a2, b2)) {  
                  
                *//Agua con bloque debajo. Se propaga hacia los lados.*  
                **long** **long** bloqueados = a1 & b2;  
                a1 |= bloqueados >> 1;  
                a1 |= bloqueados << 1;  
                  
                *//No propagar encima de bloques.*  
                a1 &= ~b1;  
                  
                *//Propagar agua hacia abajo.*  
                a2 |= a1;  
                a2 &= ~b2;  
                  
                *//Escribir linea*  
                **for** (**int** i = 0; i < n; i++){  
                        **if** (((1LL<<i)&a1) != 0) cout << *"="*;  
                        **else** **if** (((1LL<<i)&b1) != 0) cout << *"#"*;  
                        **else** cout << *"."*;  
                }  
                cout << endl;  
                  
                a1 = a2;  
                b1 = b2;  
        }  
          
        *//La ultima linea siempre será de bloques.*  
        **for** (**int** i = 0; i < n; i++) cout << *"#"*;  
  
        cout << endl << *"You are welcome, Notch!"* << endl;  
}

En un ordenador de 64 bits, este algoritmo es unas 6 veces más rápido que una solución que utiliza vectores. Nuestra idea al poner este problema era que el caso grande sólo entrara con máscaras de bits. Pero resulta que el servidor del juez es 32 bits, y los dos algoritmos eran similarmente rápidos. Normal, porque estamos usando long long, que son enteros de 64 bits, y los ordenadores de 32 bits los manejan más lento.  
  
Por esto, se puede sacar 100 puntos sin máscaras de bits, pero está bien saber que también se puede hacer así. No es un problema que las máscaras de bits optimizen mucho. Los problemas en los que de verdad son útiles son los que te ahorran usar un map<vector<bool>, algo> o alguna otra estructura de datos "complicada".