GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: OPERACIONES DE BITS

1. Introducción

El sistema de numeracion decimal (Base 10) esta compuesto por 10 digitos: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y todos los numeros estan compuestos por la combinacion de estos, por otro lado el Sistema Binario (Base 2) esta compuesto unicamente por dos digitos: 0 y 1, Bit es el acronimo de Binary digit (digito binario), entonces un bit representa uno de estos valores 0 o 1, se puede interpretar como un foco que tiene 2 estados: Encendido(1) y Apagado(0). Existen varios metodos par convertir numeros de una base a otra.

Los datos que usamos en nuestros programas, internamente, estan representados en Binario con una cadena de bits. Por ejemplo un **int** tiene 32 bits. Entonces muchas veces se requiere hacer operaciones de bits, ya sea por que estas se ejecutaran mas rápido que otras mas complejas como la multiplicacion, o por que se quiere modificarlos.

En esta guía trataremos las operaciones de bits que nos seran de mucha ayuda para la resolución de problemas de algoritmia

2. Conocimientos previos

2.1. Sistema de numeración

Un sistema de numeración es un conjunto de símbolos y reglas de generación que permiten construir todos los números válidos.

2.2. Sistema binario

El sistema binario, también llamado sistema diádico en ciencias de la computación, es un sistema de numeración en el que los números son representados utilizando únicamente dos cifras: 0 (cero) y 1 (uno). Es uno de los sistemas que se utilizan en las computadoras, debido a que estas trabajan internamente con dos niveles de voltaje (0 apagado, 1 conectado), por lo cual su sistema de numeración natural es el sistema binario.

2.3. Complemento a Dos

El complemento a 2 es una forma de representar números negativos en el sistema binario. El complemento a dos de un número N , expresado en el sistema binario con n dígitos, se define como:

$$C_2(N) = 2^n - N$$

donde total de números positivos será $2^{n-1} - 1$ y el de negativos 2^{n-1}, siendo n el número de bits. El 0 contaría como positivo, ya que los positivos son los que empiezan por 0 y los negativos los que empiezan por 1.

3. Desarrollo

3.1. Representación bit

En programación, un entero de n bits se almacena internamente como un número binario que consta de n bits. Por ejemplo, el tipo int de C++ es un tipo de 32 bits, lo que significa que cada número int consta de 32 bits.

Aquí está la representación de bits del número int 43:

$$00000000000000000000000000101011$$

Los bits de la representación están indexados de derecha a izquierda. Para convertir una representación de bits $b_k \ldots b_2 b_1 b_0$ en un número, podemos usar la fórmula:

$$b_k 2^k + \cdots + b_2 2^2 + b_1 2^1 + b_0 2^0.$$

Por ejemplo,

$$1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 43$$

La representación en bits de un número puede tener **signo o no**. Normalmente se utiliza una representación con signo, lo que significa que se pueden representar números tanto negativos como positivos. Una variable con signo de n bits puede contener cualquier número entero entre -2^{n-1} y $2^{n-1}-1$. Por ejemplo, el tipo int en C++ es un tipo con signo, por lo que una variable int puede contener cualquier número entero entre -2^{31} y $2^{31}-1$.

El primer bit en una representación con signo es el signo del número (0 para números no negativos y 1 para números negativos), y los $n-1$ bits restantes contienen la magnitud del número. Se utiliza el **complemento a dos**, lo que significa que el número opuesto de un número se calcula invirtiendo primero todos los bits del número y luego aumentando el número en uno.

Por ejemplo, la representación en bits del número int -43 es:

$$11111111111111111111111111010101$$

En una representación sin signo, sólo se pueden utilizar números no negativos, pero el límite superior de los valores es mayor. Una variable sin signo de n bits puede contener cualquier número entero entre 0 y 2^n-1. Por ejemplo, en C++, una variable int sin signo puede contener cualquier número entero entre 0 y $2^{32}-1$.

Existe una conexión entre las representaciones: un número con signo $-x$ es igual a un número sin signo $2^n - x$. Por ejemplo, el siguiente código muestra que el número con signo $x = -43$ es igual al número sin signo $y = 2^{32} - 43$:

```
int x = -43;
unsigned int y = x;
cout<< x << "\n"; // -43
cout<< y << "\n"; // 4294967253
```

Si un número es mayor que el límite superior de la representación de bits, el número se desbordará. En una representación con signo, el siguiente número después de $2^{n-1}-1$ es -2^{n-1}, y en una representación sin signo, el siguiente número después de 2^n-1 es 0. Por ejemplo, considere el siguiente código:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Inicialmente, el valor de x es $2^{31}-1$. Este es el valor más grande que se puede almacenar en una variable int, por lo que el siguiente número después de $2^{31}-1$ es -2^{31}.

3.2. Operaciones bit a bit

Son operaciones logicas que se ejecutan sobre bits individuales.

3.2.1. AND

El AND bit a bit, toma dos números enteros y realiza la operación AND lógica en cada par correspondiente de bits. El resultado en cada posición es 1 si el bit correspondiente de los dos operandos es 1, y 0 de lo contrario. La operacion AND se representa con el signo $\&$. Por ejemplo, $22\&26 = 18$, porque:

$$\begin{array}{cll} & 10110 & (22) \\ \& & 11010 & (26) \\ \hline = & 10010 & (18) \end{array}$$

3.2.2. OR

Una operación OR de bit a bit, toma dos números enteros y realiza la operación OR inclusivo en cada par correspondiente de bits. El resultado en cada posición es 0 si el bit correspondiente de los dos operandos es 0, y 1 de lo contrario. La operacion OR se representa con el signo $|$. Por ejemplo, $22|26 = 30$, porque:

$$\begin{array}{cll} & 10110 & (22) \\ | & 11010 & (26) \\ \hline = & 11110 & (30) \end{array}$$

3.2.3. NOT

El NOT bit a bit, es una operación unaria que realiza la negación lógica en cada bit, invirtiendo los bits del número, de tal manera que los ceros se convierten en 1 y viceversa.

La operación not $\sim x$ produce un número donde todos los bits de x se han invertido. La fórmula $\sim x = -x - 1$ se cumple, por ejemplo, $\sim 29 = -30$.

El resultado de la operación not a nivel de bits depende de la longitud de la representación de bits, porque la operación invierte todos los bits. Por ejemplo, si el Los números son números int de 32 bits, el resultado es el siguiente:

$$\begin{array}{rcrl} x & = & 29 & 00000000000000000000000000011101 \\ \sim x & = & -30 & 11111111111111111111111111100010 \end{array}$$

3.2.4. XOR

El XOR bit a bit, toma dos números enteros y realiza la operación OR exclusivo en cada par correspondiente de bits. El resultado en cada posición es 1 si el par de bits son diferentes y 0 si el par de bits son iguales. La operacion XOR se representa con el signo $\wedge$. Por ejemplo, $22 \wedge 26 = 12$, porque:

$$\begin{array}{rll} & 10110 & (22) \\ \wedge & 11010 & (26) \\ \hline = & 01100 & (12) \end{array}$$

3.3. Operaciones de Desplazamiento

Otra operacion importante es la de desplazar bits, se tiene 2 movientos: El desplazamiento a la derecha y el desplazamiento a la izquierda. Al desplazar los bits los espacios faltantes son rellenados con ceros.

3.3.1. Desplazamiento a la izquierda *Left Shift*

El desplazamiento a la izquierda (*left shift*) es la operacion de mover todos los bits una cantidad determinada de espacion hacia la izquierda. Esta operacion esta representado por los signos $<<$. Por cada desplazamiento a la izquierda se agregara un cero a la derecha de todo el numero.

3.3.2. Desplazamiento a la derecha *Rigth Shift*

El desplazamiento a la derecha (*rigth shift*) es la operacion de mover todos los bits una cantidad determinada de espacios hacia la derecha. Esta operacion esta representada por los signos $>>$. Al dezplazar el número hacia la derecha los bits menos significativos (los ultimos) se perderán.

4. Implementación

4.1. C++

El compilador g++ proporciona las siguientes funciones para contar bits:

- **__builtin_clz(x):** El número de zeros que comienza el número.
- **__builtin_ctz(x):** El número de zeros que termina el número.
- **__builtin_popcount(x):** El número de unos tiene el número en su representación binaria.
- **__builtin_parity(x):** La paridad (par o impar) del número de unos

Las funciones se pueden utilizar de la siguiente manera:

```
int x = 5328; // 00000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Si bien las funciones anteriores solo admiten números **int**, también hay versiones **long long** de las funciones disponibles con el sufijo ll.

5. Aplicaciones

Con todas las operaciones anteriormente vistas podemos hacer muchas cosas interesantes a la hora de programar, ahora les mostraremos algunas de las mas importante aplicaciones de estas operaciones:

1. **Estado de un bit:** Con el uso de las operaciones AND y left shift podemos determinar el estado de un bit determinado. Por ejemplo: Supongamos que tenemos el numero 17 y queremos saber si su quinto bit esta encendido, lo que haremos sera desplazar cuatro posiciones el número 1, notese que se desplaza $n-1$ veces los bits, y realizamos la operacion AND si el resultado es diferente de 0 el bit esta encendido, por el contrario esta apagado.
2. **Apagar un bit:** Usando las operaciones AND, NOT y left shift podemos apagar un determinado bit. Por ejemplo: Supogamos que tenemos el número 15 y queremos apagar su segundo bit, lo que haremos sera desplazar una posición el número 1, aplicamos NOT a este número y luego AND entre ambos, con esto habremos apagado el segundo bit del numero 15.
3. **Encender un bit:** Usando las operaciones OR y left shift encenderemos un bit determinado de un número. Por ejemplo: Supongamos que tenemos el número 21 y queremos encender su cuarto bit, lo que haremos sera desplazar tres posiciones el número 1, y realizamos la operación OR entre ambos números.
4. **Multiplicación y División por** 2**:** Una forma rápida de multilicar o dividir un número por 2 es haciendo uso del desplazamiento de bits, pues si tenemos un número entero n, y lo

desplazamos una pocición a la derecha el número se dividira entre 2 con resultado entero $(n/2)$ y si por el contrario desplazamos el número una posición a la izquierda el número se multiplicará por 2 $(2 \times n)$.

5. **Dos elevado a la n:** Si tenemos el número 1 y lo desplazamos n veces a la izquierda obtendremos como resultado 2^n .
6. **Máscara de bits o *BitMask*:** es un algoritmo sencillo que se utiliza para calcular todos los subconjuntos de un conjunto.

Las operaciones de bits proporcionan una manera eficiente y conveniente de implementar algoritmos de programación dinámica cuyos estados contienen subconjuntos de elementos, porque dichos estados pueden almacenarse como números enteros.

6. Complejidad

Muchos algoritmos se pueden optimizar mediante operaciones de bits. Estas optimizaciones no cambian la complejidad temporal del algoritmo, pero pueden tener un gran impacto en el tiempo de ejecución real del código.

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver con operaciones de bits

- DMOJ - Potencia Menor