Manejo de Binarios y Mascaras (Bitwise)

Por Ariel Parra

[Γα=Ω5]

Números Binarios

En los numeros conformados por bits el termino de hasta la derecha se le conoce como el bit menos significativo (Least Significant Bit, "LSB") y el de hasta la izquierda es el bit más significativo (Most Significant Bit, "MSB").

También existen números negativos donde el MSB dicta el signo 1 para negativo y 0 para positivo, aunque esto solo significa que el MSB sera un némero negativo al que se le suman los demas números, por ejemplo en la serie de 8 bits "10000000" el 1 significa que esta negativo y esta en la octava posición por lo que este seria 2^8 = -128, con un byte con signo se puede ir desde -128 a 127 ya que pasamos por el cero. la forma de tener un valor sin signo en c es usando el tipo de dato `unsigned` el cual descarta el bit de signo, dandonos obteninendo una mayot maginutud del dato.

¿Qué es el Bit Masking?

Es el proceso de modificación y utilización de representaciones binarias de números o cualquier otro dato se conoce como bitmasking. Usando una máscara, múltiples bits en un byte, word, etc. pueden ser estar encendidos o apagado, o también puede ser invertido de encendido a apagado en un solo bit.

Operadores de manipulación de bits (bitwise operators) en C

Símbolo	Operador
&	bitwise AND
|	bitwise inclusive OR
^	bitwise XOR (exclusive OR)
<<	left shift
>>	right shift
~	bitwise NOT (unario)

Todas estas operaciones tienen complejidad de O(1)

Son muy similares a los operadores booleanos, pero no se deben confundir con dichos operadores.

Bitwise	Logico	Bitwise	Logico
a & b	a && b	a ^ b	a != b
a | b	a || b	~a	!a

Ejemplos de operaciones bitwise:

```
  11001000          11001000          11001000          11001000
& 10111000        | 11111000        ^ 11001001        ~
----------        ----------        ----------        ----------
= 10001000        = 11111000        = 00000001        = 00110111
```

What are Bit Masks, and how do I use them? (examples in C)

Tabla de verdad

bit a	bit b	a & b (a AND b)	a | b (a OR b)	a ^ b (a XOR b)	~a (NOT a)
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operadores de asignación bitwise

EL asignar valores optimiza el codigo al no crear copias de la misma vairable y por ende optimiza el espacio auxiliar a O(1), veremos el ejemplo con la variable: `int val=0b11001000;`

Symbol	Operator	Ejemplo
&=	bitwise AND assignment	val &= 0b10111000;
|=	bitwise inclusive OR assignment	val |= 0b11001000;
^=	bitwise exclusive OR assignment	val ^= 0b11001000;
=~	bitwise NOT (unario)	val = ~val;
<<=	left shift assignment	val <<= 1;
>>=	right shift assignment	val >>= 2;

Algoritmos optimizados por el uso de bitmasking

Algoritmo de un numero decimal `int n` a un vector binario `vector<int> vecBin;`:

```
while (n > 0) {
    vecBin.push_back(n % 2); //se guarda el LSB
    n /= 2;//n = n / 2;
} // O(logn) & Space: O(1)
```

```
while (n > 0) {
    vecBin.push_back(n & 1); //LSB
    n >>= 1;// (n = n / (2^1))
} // O(1) & Space: O(1)
```

```
for (int i = 0; n > 0 ; ++i) // O(1) & Space: O(1)
    vecBin.push_back( (n >> i) & 1 );  //directo
```

```cpp
void swap(int &a, int &b) { // O(1) & Space: O(1)
    if (a == b) return;
    a = a ^ b; // a ^= b;
    b = a ^ b;
    a = a ^ b; // a ^= b;
}
```

```cpp
const int SIZE_INT = sizeof(int) * 8 - 1; // MSB 31 (0 a 31 son 32 bits)

int bit_max(int &a, int &b) { // O(1) & Space: O(1)
    return a - ((a - b) & ( (x - b) >> (SIZE_INT) );
}

int bit_min(int &a, int &b) { // O(1) & Space: O(1)
    return ((a - b) & ( (a - b) >> (SIZE_INT) ) ) + b;
}
```

`num&1` VS `num%2!=0`

Verificar si un número es impar usando `num & 1` suele ser más eficiente que usar num `% 2 != 0` debido a la diferencia en cómo se realizan estas operaciones a nivel del CPU, aunque ambas tengan una complejidad O(1).

```
AND AL, 1       ; AL = num & 1 (2 ciclos CPU)
CMP AL, 1       ; Comparar con 1 (1 ciclo CPU)
JE EsImpar      ; Si es igual a 1, el número es impar (1 ciclo CPU)
```

```
MOV AX, num     ; Mover el número a AX (1 ciclo CPU)
MOV BX, 2       ; Divisor es 2 (1 ciclos CPU)
DIV BX          ; Dividir AX por BX, AL = residuo (10 a 20 ciclos CPU)
CMP AL, 0       ; Comparar el residuo con 0 ((1 ciclo CPU))
JNE EsImpar     ; Si no es igual a 0, es impar (1 ciclos CPU)
```

Ejercicio de Clase

Conforme a los operadores de bitwise aprendidos y operadores aritmeticos regulares. Realizar una función en c++ que multiplique por 10 a cualquier número

```
int xTen(int &n) {
    return; //código
}
```

Solución

```
int xTen(int &n) {
    return n<<3 + n<<1; // n*(2^3) + n*(2^1) = n*8 + n*2 = n*(8+2) = n*10
}
```

Operaciones con Conjuntos (subsets)

Operación	Notación de Conjuntos	Notación de Bits
Intersección	a ∩ b	a & b
Unión	a ∪ b	a | b
Complemento	ā	~a
Diferencia	a \ b	a&(~b)

El código construye los conjuntos $x = \{1, 3, 4, 8\}$ y $y = \{3, 6, 8, 9\}$, y luego construye el conjunto $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Iteración a través de Subconjuntos

Recorrer todos los subconjuntos de {0, 1, ..., n - 1}:

```
for (int b = 0; b < (1<<n); ++b)
    // procesar el subconjunto b
```

Recorrer subconjuntos con exactamente `k` elementos:

```
for (int b = 0; b < (1<<n); ++b)
    if (__builtin_popcount(b) == k)
        // procesar el subconjunto b
```

Recorrer los subconjuntos de un conjunto `x`:

```
int b = 0;
do {
    // procesar el subconjunto b
} while (b = (b - x) & x);
```

Bitsets

```
bitset<8> decBset(8); bitset<8> strBset("1100"); bitset<8> binBset(0b001);

decBset.set(4);         // Set the bit at idx 4 to 1
decBset.reset(4);       // Reset the bit at idx 4 to 0
decBset.flip(0);        // Flip the bit at idx 0 (0 becomes 1, and 1 becomes 0)

int numSetBits = decBset.count();  // Count the number of bits that are set to 1
bool bit2IsSet = decBset.test(2);  // Check if the bit at idx 2 is set to 1
bool anyBSet = decBset.any();       // Check if any bit is set to 1
bool noBSet = decBset.none();       // Check if no bits are set to 1
bool allBSet = decBset.all();       // Check if all bits are set to 1
int bsetSize = decBset.size();
string bsetStr = decBset.to_string();
unsigned long bsetULong = decBset.to_ulong();
unsigned long long bsetULLong = decBset.to_ullong();
```

GCC __builin functions

Las funciones `__builtin` de GCC proporcionan operaciones de bajo nivel optimizadas para trabajar con bits:

```
__builtin_popcount(x); // Cuenta el número de bits en 1 (bits sets)
```

```
__builtin_parity(x); /* Verifica la paridad de un número. Devuelve verdadero (1)
 si el número tiene paridad impar (número impar de bits establecidos), de lo contrario devuelve falso (0) */
```

```
__builtin_clz(x); // Cuenta el número de ceros iniciales (Count Leading Zeros)
```

```
__builtin_ctz(x); // Cuenta el número de ceros finales (Count Trailing Zeros)
```

```
__builtin_ffs(x); // (Find First Set) devuelve el índice del bit menos significativo establecido en x+1
```

```
__lg(x); // Devuelve el índice del bit más significativo establecido
```

Problemas

- **1805A** We Need the Zero ⮭
- **1527A** And Then There Were K ⮭

Referencias

- Back To Back SWE. (2019). *Add Two Numbers Without The "+" Sign (Bit Shifting Basics)* [video]. Recuperado de https://youtu.be/qq64FrA2UXQ?si=lQCA5ATPIU7N6u3o
- Bisht, L. (2023). *What is Bitmasking*. Recuperado de https://www.geeksforgeeks.org/what-is-bitmasking/
- Bora, S. (2023). *BITMASKS — FOR BEGINNERS*. Recuperado de https://codeforces.com/blog/entry/18169
- FSF. (2024). *6.63 Other Built-in Functions Provided by GCC*. Recuperado de https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html
- GeeksforGeeks. (2023). *Builtin functions of GCC compiler*. Recuperado de https://www.geeksforgeeks.org/builtin-functions-gcc-compiler/
- GeeksforGeeks. (2023). *Convert Binary to Decimal in C*. Recuperado de https://www.geeksforgeeks.org/c-binary-to-decimal/

- GeeksforGeeks. (2024). *Program for Decimal to Binary Conversion*. Recuperado de https://www.geeksforgeeks.org/program-decimal-binary-conversion/
- Golovanov, A. (2020). *C++ tips and tricks*. Recuperado de https://codeforces.com/blog/entry/74684
- Jacob Sorber. (2019). *What are Bit Masks, and how do I use them? (examples in C)* [video]. Recuperado de https://youtu.be/Ew2QnDeTCCE?si=eEFD36IDGyuz6O7Q
- Kalita, R. (2022). *Bit Masking*. Recuperado de https://www.scaler.com/topics/data-structures/bit-masking/
- Kumar, A. (2023). *Bit Tricks for Competitive Programming*. Recuperado de https://www.geeksforgeeks.org/bit-tricks-competitive-programming/
- Laaksonen, A. (2018). *Competitive Programmer's Handbook*. Recuperado de https://cses.fi/book/book.pdf
- Yousefi, H. (2015). *C++ Tricks*. Recuperado de https://codeforces.com/blog/entry/15643