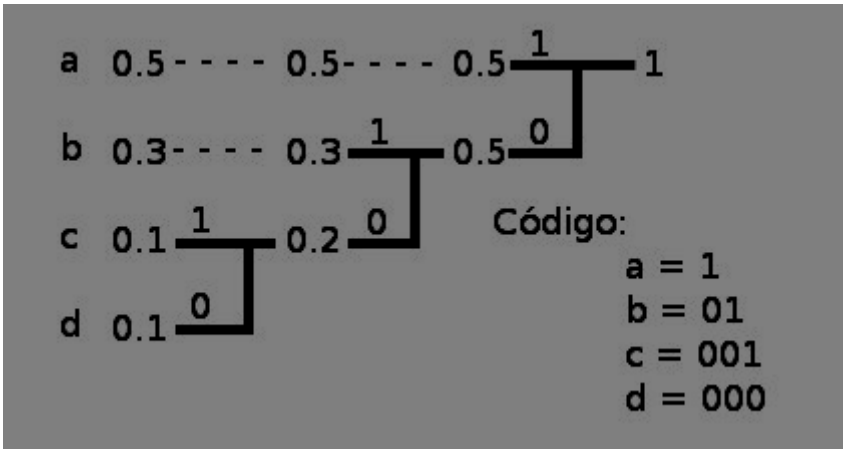
Teoría de la Información

Operaciones a nivel bit

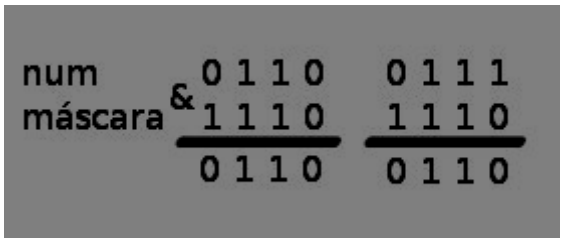
En primer lugar veamos la necesidad de trabajar a nivel bit, y creo que la mejor forma es dando un ejemplo práctico.
Supongamos que obtuvimos un código de Huffman sobre unos caracteres y nos dio la siguiente codificación:



Entonces, por ejemplo la siguiente cadena “aacbc”, la codifico y produce la secuencia 1100101001.
Ahora bien, si uno representara cada bit como un char o un integer estaríamos expandiendo siempre el archivo original y esto no es lo que queremos.
Entonces introduciremos y refrescaremos conceptos de lógica binaria y exactamente para que sirve cada operador, para luego ver implementaciones de algoritmos en Java.

Lógica Binaria y aplicaciones

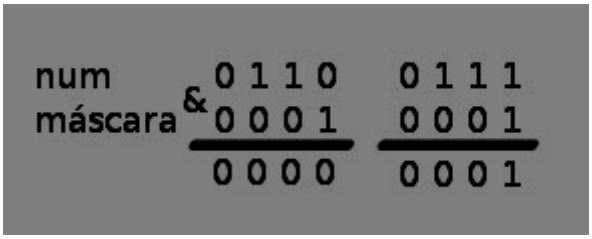
AND (&)
Es útil para “eliminar” bits de un operando, ya que al hacer **num** & **máscara** (con un 0 en la posición deseada de mascara) el resultado será siempre 0 en esa posición, sin importar el valor que tuviera **num** en ese lugar.
Ejemplo:



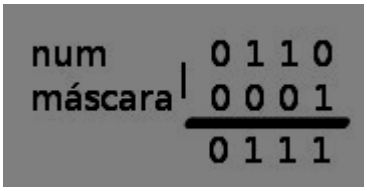
La idea del segundo ejemplo es que el último bit en 0, sea cual fuera el valor de ese bit en **num** (por eso la **máscara** es 1110, los “1” hacen que esas posiciones no se alteren en el resultado.
El AND también es útil para “averiguar” el valor del bit en una posición, ya que si se hace un AND con **máscara** =”1”, el resultado es “1” solo si el bit de **num** es “1” en esa posición, y es “0” si el bit original es “0”.

Ejemplo:

Teoría de la Información



OR inclusivo (|)
En este caso, si se hace OR con **máscara** "1" hace que el resultado sea "1" siempre (no importa si en **num** tenemos "1" o "0") y si se hace con **máscara** "0" el bit original se deja sin cambio.
Ejemplo:

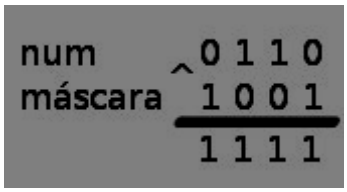


En este ejemplo se ve como se forzó a ser "1" al primer bit y dejar los restantes como antes.

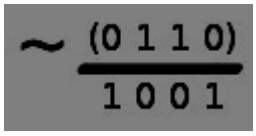
OR exclusivo (^)

Es útil para generar el valor complementario de cualquier bit individual en una variable, ya que al operar con un "1" en la **máscara**, hace que ese bit tome el valor opuesto en esa posición.

Ejemplo:



Complemento (~)
La utilidad de esta operación es invertir los bits.
Ejemplo:



Desplazamiento
<< η Desplaza todos los bits η posiciones a izquierda.
>> η Desplaza todos los bits η posiciones a derecha.

Vale aclarar que η nunca puede ser negativo, ya que daría error.
Cada vez que se hace un desplazamiento se completa con ceros en el otro lado (no se trata de una rotación).
En el caso de un valor con signo, como se representa con un "1" en el bit de orden superior, cada desplazamiento a derecha se introduce un 1 por izquierda.
Ejemplo:

Teoría de la Información

0 0 1 1 << 2 -> 1 1 0 0 >> 1 -> 0 1 1 0

Cada desplazamiento a izquierda, es una multiplicación por 2.
Cada desplazamiento a derecha, es una división por 2.

La ventaja de los corrimientos es que son menos costosos que hacer las operaciones, de multiplicación y división.

Manejo de bits en C/C++ y Java

Básicamente el manejo de bits en Java es similar al de C/C++.
Acá tenemos una tabla con cada operador, como usarlo y que es lo que hace.

Operador	Utilización	Resultado
<<	A << B	Desplazamiento de A a la izquierda en B posiciones
>>	A >> B	Desplazamiento de A a la derecha en B posiciones, tiene en cuenta el signo.
>>>	A >>> B	Desplazamiento de A a la derecha en B posiciones, no tiene en cuenta el signo.
&	A & B	Operación AND a nivel de bits
|	A | B	Operación OR a nivel de bits
^	A ^ B	Operación XOR a nivel de bits
~	~A	Complemento de A a nivel de bits

Todas estas operaciones son útiles en procesamiento de señales, para codificación, compresión y encriptación de la información.

Ejemplo:

El siguiente programa considera un número y lo procesa, “descomponiendo” sus dígitos binarios por medio de operaciones AND y corrimientos <<.

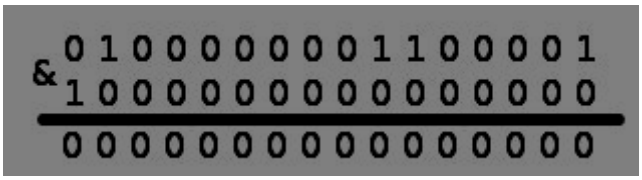
```
public void generaBits(char num){
        char mascara = 1 <<  15;//desplaza el 1, 15 lugares a la izq (32768)
        for(int i = 0; i<16; i++){
                if((num & mascara)==32768) //si el 1º bit de num es 1
                        System.out.print("1");
                else
                        System.out.print("0");
                num = (char) (num << 1);
        }
}
```

Teoría de la Información

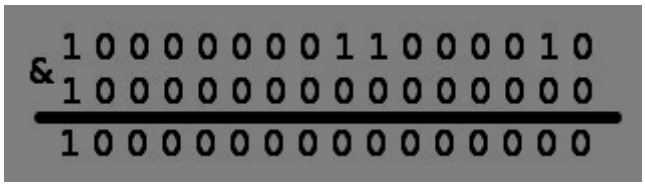
Seguimiento:

num: 0100000001100001
máscara: 1000000000000000

i=0)

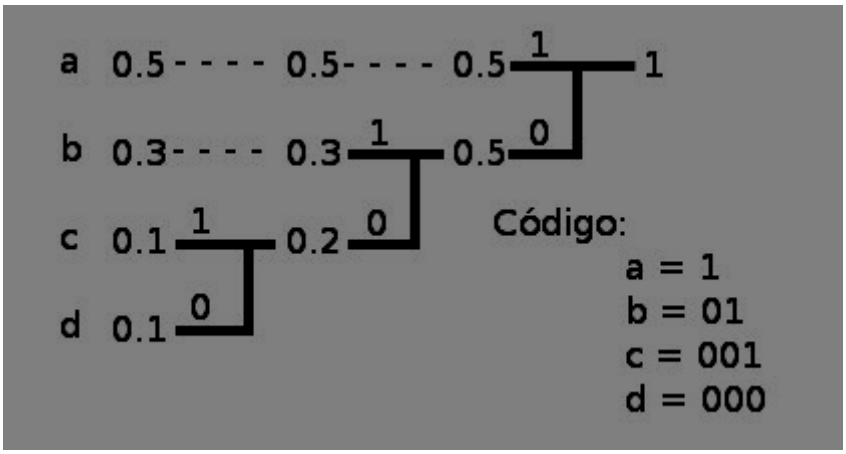


i=1)



¿Qué sucede en codificación?

Volviendo al ejemplo del principio, cuando queremos generar un archivo comprimido tenemos que operar a nivel bit.
Ejemplo:
Código de Huffman produce código binario para cada símbolo.
Recordando la codificación del ejemplo:



Entonces, la idea es:
a) Procesar cada símbolo a codificar.
b) Recuperar su codificación como secuencia de bits.
c) “Empaquetar” los bits en un char (16 bits en Java,8 bits en C++), en un short, en un int, etc. Haciendo tantos corrimientos (<<) como dígitos tenga el código y hacer un OR (|) si el bit es 1, e ir llevando la cuenta de los bits procesados. Cuando se completa, se lleva al archivo.

```
private void generarCodificacion(){
                buffer=0;
                cant_digitos=0;
                while(!fin(entrada){
                        simbolo = getSimbolo();
                        codigo = getCodigo(simbolo);
                        n =longitud(codigo);
                        while(n>0){
                                buffer=buffer<<1;
                                if(siguiente(codigo)=='1'){
                                        buffer = buffer|1;
                                }
                                cant_digitos++;
                                if(cant_digitos==16){
                                        escribir(buffer);
                                        buffer=0;
                                        cant_digitos=0;
                                }
                                n--;
                        }
                }
                if ((cant_digitos<16)&&(cant_digitos!=0)){
                   buffer=buffer<<(16-cant_digitos);
                   escribir(buffer);
                }
        }
```