Ficha 12

Gestión de Bits

1.) Correspondencia entre binario, hexadecimal y base 10.

El lenguaje Java provee un amplio conjunto de operadores para manipulación bits. Si bien en la mayor parte de los casos un programador puede manejarse con variables de tipo primitivo que agrupan entre uno y ocho bytes para representar un valor, no es menos cierto que muchas aplicaciones exigen que el programador realice alguna manipulación de los bits contenidos en una variable de cualquier tipo. Un ejemplo es el desarrollo de un compresor de datos: el archivo contiene bytes que en última instancia puede asumirse que representan valores ASCII de 8 bits, pero para lograr una menor ocupación de espacio esos bytes deben reemplazarse por otros que contengan representaciones de códigos de Huffman... y eso requiere manipulación de bits.

Quizás el primer problema que debe enfrentar un programador que necesite manipular los bits de una variable, es el de poder representar adecuadamente una secuencia binaria y asignar tal secuencia en una variable. Este no es un tema menor: escrito en binario, un número que en base 10 sería "pequeño" podría ocupar muchos bits, y eso haría impráctico el manejo de tal representación en un programa.

Para evitar esta clase de problemas, se usan bases que son *potencia de dos*, entre ellas la base 16 (*hexadecimal*) y la base 8 (*octal*). La ventaja de una base potencia de dos, es que cada dígito de esa base ocupa un *número fijo* de tantos bits como indica la potencia de la base 2: en base 16 (que es 2^4) cada uno de los 16 digitos (*0* al *9* más *A*, *B*, *C*, *D*, *E*, *F*) se puede representar con 4 bits. Y en base 8 (que es 2^3) cada uno de los 8 dígitos (*0* al *7*) se puede representar con 3 bits. Eso implica que la conversión de binario a esas bases (y viceversa) se puede hacer por *reemplazo directo*: cada grupo de *n* bits (siendo *n* la potencia de la base) se reemplaza por el dígito equivalente en esa base... sólo se requiere la tabla de conversión (sin tener que hacer cálculos de potencias o de restos).

Ejemplo: la siguiente secuencia binaria de 16 bits: 0010100011110001 puede escribirse directamente como 28F1 en hexadecimal, pues las subsecuencias {0010, 1000, 1111, 0001} equivalen respectivamente a los dígitos hexadecimales {2, 8, F, 1}. Así, si se requiere que una variable adquiera el valor representado por esa secuencia binaria, el programador sólo debe convertirla a hexadecimal y asignar la conversión.

En la práctica, la base hexadecimal (en adelante, *hexa*) es la más cómoda y la más usada: como se vió, un dígito hexa equivale a 4 bits, lo cual es medio byte. Dos dígitos hexa juntos hacen un byte, y así resulta cómodo representar cualquier byte en forma de pares de valores hexa. Nunca está de más, entonces, contar con la conocida tabla de conversión de bases (en la cual mostramos los 16 dígitos hexa, con su conversión a binario y su correspondencia en base 10):

Valor binario	Dígito hexa	Valor en base 10
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4

0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

La siguiente cuestión es cómo hacer en Java para asignar en una variable un valor hexa (u octal llegado el caso...) Cuando un programador necesita operar a nivel de bits, lo común es que tenga en mente un "patrón de bits" y no tanto el valor en base 10 que ese patrón significa... Dijimos que es simple pasar el patrón a hexa y asignar el resultado, pero ¿cómo se trabaja con valores hexa u octal?

En Java, cualquier valor entero que comience con el prefijo *0* (*cero*) se toma como expresado en octal, y cualquier entero que comience con el prefijo *0x* (*cero equis*) se toma como expresado en hexa. El valor es un entero como cualquier otro, pero expresado en otra base en lugar de base 10. Las siguientes asignaciones son válidas (las letras hexa pueden ir en mayúscula o minúscula, indistintamente):

```
short a = 0x28f1;    // hexa:  asigna la secuencia 0010100011110001
short b = 0341;      // octal:  asigna la secuencia 011100001
```

Si se muestran los valores de las variables con *System.out.print()*, se verá que en el primer caso la variable *a* queda valiendo el valor 10481 (en base 10), mientras que *b* queda valiendo 225 (en base 10). Es decir que las instrucciones anteriores hubieran sido totalmente equivalentes a estas otras:

```
short a = 10481;
short b = 225;
```

Recordemos que un programador que trabaja a nivel de bits, muchas veces no sabe ni necesita saber cual es el valor en base 10: sólo opera con el patrón de bits que se ajusta a la operación que quiere llevar a cabo. Y en ese contexto, la representación en hexa es mucho más cómoda que la representación en base 10.

Notemos que los métodos de salida por consola en Java, muestran *siempre en base 10* el valor de una variable numérica, sin importar si el valor se asignó en base 10, hexa u octal. La variable mostrada siempre almacena el valor en binario, sea como sea que se haya hecho la asignación; y *System.out.print()* o *System.out.println()* convierten el valor a base 10 antes de mostrarlo... Así, la secuencia:

```
short a = 0x28f1;
short b = 0341;
System.out.println("a: " + a);
System.out.println("b: " + b);
```

mostrará en la consola de salida algo como:

```
a: 10481
```

b: 225

Es también útil notar que la clase *java.lang.Integer* provee numerosos métodos (la mayor parte *static*) para obtener cadenas que representan un número entero en distintas bases. La siguiente tabla es un extracto del javadoc de la clase para los métodos citados:

static String	**toBinaryString**(int i) Retorna una representación como *String* del parámetro, tal que la cadena queda como un valor sin signo en base 2.
static String	**toHexString**(int i) Retorna una representación como *String* del parámetro, como un entero sin signo en base 16.
static String	**toOctalString**(int i) Retorna una representación como *String* del parámetro, como un entero sin signo en base 8.
String	**toString**() Retorna un *String* que representa al valor *Integer*.
static String	**toString**(int i) Retorna un *String* que representa al valor tomado como parámetro.
static String	**toString**(int i, int radix) Retorna un *String* que representa al primer parámetro, en la base especificada por el segundo parámetro.
static int	**parseInt**(String s, int radix) Retorna un valor *int* en base 10, que representa el valor extraido desde la cadena *s*, suponiendo que la cadena viene en la base indicada por el segundo parámetro. Si la conversión no es posible, lanza una *NumberFormatException*.

Los siguientes ejemplos (extraidos del modelo que acompaña a esta Ficha) muestran la forma de uso de algunos de estos métodos. Note que el método *toString(int i, int radix)* puede ser usado en reemplazo de cualquiera de los otros tres mostrados, usando *radix 2*, *8* y *16* respectivamente. También se puede usar esta versión de *toString()* para obtener una cadena en cualquier base (en el ejemplo, se obtiene una cadena que representa al valor en base 13):

```
a = 21439;  // un número cualquiera en base 10
System.out.println("Valor en base 10: " + a);
System.out.println("Valor en base 2:  " +  Integer.toBinaryString(a));
System.out.println("Valor en base 8:  " +  Integer.toOctalString(a));
System.out.println("Valor en base 16: " + Integer.toHexString(a));
System.out.println("Valor en base 13: " + Integer.toString(a, 13));
```

El resultado de correr este segmento es la siguiente salida por consola estándar:

```
Valor en base 10: 21439
Valor en base 2:  101001110111111
Valor en base 8:  51677
Valor en base 16: 53bf
```

Valor en base 13: 99b2

Un detalle importante a considerar es que el compilador *no permitirá* asignar valores que excedan el rango de una variable, aún cuando el valor venga en octal o en hexa. Los siguientes ejemplos no compilan (todos los valores asignados son mayores a 127, que es el límite superior de los valores que puede tomar una variable *byte*):

```
byte x = 200;      // no compila: 200 es mayor a 127
byte y = 0777;     // no compila: 0777 en octal es 511 en base 10, y es mayor a 127
byte z = 0xF1;     // no compila: 0xF1 en hexa es 241 en base 10, y es mayor a 127
```

De hecho, el valor *0xF1* en binario sería *11110001*. Suponga que un programador necesita almacenar esa secuencia en una variable. Pero si lo intenta con una variable *byte* no compilará. ¿Qué debe hacer entonces? Puede usar una variable *short*, y llenar con ceros el primer byte:

```
short z = 0x00F1;          // asigna 00000000 11110001
```

Este tipo de trucos son comunes para evitar el error de compilación resultante de asignar valores fuera de rango. Si una variable mide más de un byte, entonces el de más a la derecha se dice el byte *menos significativo*, y el de más a la izquierda es el *más significativo*. En el ejemplo anterior, la asignación provoca que el byte más significativo se complete con ocho ceros, mientras que la secuencia 11110001 queda almacenada en el menos significativo. Obviamente, el mismo resultado se logra asignando directamente el valor 0xF1 (el byte más significativo se completa con ceros de todos modos) o asignando el valor 241:

```
z = 0xF1;          // asigna 00000000 11110001
z = 241;           // asigna 00000000 11110001
```

2.) Números enteros negativos en binario.

La representación de números enteros en un computador se hace llevando ese número a base 2, y almacenando la secuencia de bits resultante en las variables que correspondan. En Java esas variables pueden ser *byte*, *short*, *int* o *long* con 1, 2, 4 y 8 bytes respectivamente. Pero en esa representación también debe darse cabida a los números negativos. Si todos los bits de una variable se usan para representar la magnitud del número en binario, entonces todos los números representados serían positivos (tendríamos lo que en otros lenguajes se llaman enteros *sin signo*).

El hecho es que en Java todos los tipos numéricos son *con signo*: los números negativos forman indefectiblemente parte del conjunto de valores representado por cada tipo. ¿Cómo se representa un negativo en binario? Indudablemente, el problema inicial es cómo representar el propio signo menos (-), que corrientemente representamos con un guión. Pero las computadoras sólo saben de ceros y unos, y el guión no es un signo válido en ese *discreto* mundo... ¿Cómo indicar que la secuencia de bits almacenada representa un valor negativo? La solución clásica es que se use el primer bit de la izquierda (el bit más significativo) para representar el signo: si ese primer bit es cero, el número es positivo. Y si es uno, el número es negativo. Por lo tanto, en una variable *byte* son sólo siete los bits que pueden usarse para representar el valor absoluto del número... y eso restringe el valor máximo a 127:

```
byte n = 127;          ⇨          n = 0111 1111
```

Si se muestra el valor de *n* en consola de salida, no aparecerá el cero inicial (como siempre que hay un cero delante de un entero), pero note que la salida consta de siete bits... el octavo (ausente) es el primero de la izquierda, que vale cero.

Ahora bien: los números negativos no se representan sin más con un uno delante y el resto de los bits representando al valor. El número –5 no es 10000101 en binario... La representación de negativos en binario se hace con sistemas alternativos, entre los cuales los dos más conocidos son el de *notación en exceso* y el de *complemento a dos*. De estos dos, el más común a su vez es el segundo.

En el sistema de complemento a dos, la forma básica de obtener la secuencia binaria de un negativo es simple: escriba en binario el número tal y como lo haría si fuera positivo, pero empleando todos los bits que necesite para ocupar toda la precisión de la variable empleada (complete con ceros adelante si fuera necesario). Luego obtenga el complemento de esa secuencia: convierta en ceros todos los unos, y en unos todos los ceros. Finalmente, a la secuencia binaria obtenida, súmele uno. Lo que obtenga, es la representación binaria del valor inicial. Para el caso del –5, el proceso sería:

- Escribir (–5) en binario, en precisión de un byte:

  - paso 1: escribir 5 en binario, en un byte: 0000 0101
  - paso 2: complemente la secuencia: 1111 1010
  - paso 3: sume 1 a la secuencia obtenida: 1111 1011 ⇨ (-5)

Note que el proceso es *reversible*: si se parte de la representación binaria de un negativo en complemento a dos, y se aplica el mismo proceso, se obtiene la representación del número pero en positivo:

- Partir de (–5) en binario, y regresar a (5) en precisión de un byte:

  - paso 1: escribir (-5) en binario, en un byte: 1111 1011
  - paso 2: complemente la secuencia: 0000 0100
  - paso 3: sume 1 a la secuencia obtenida: 0000 0101 ⇨ (5)

Uno de los motivos por los cuales el sistema de complemento a dos es tan usado (y el motivo principal por el cual existe...) es que permite hacer sumas en forma directa, sin importar el signo de los términos: en otras palabras, tanto da que se esté haciendo una suma o una resta, el proceso es el mismo: se suman los bits en binario, y si hay acarreo se "lleva" el bit que sobre hacia la izquierda, hasta completar la suma de todos los bits. Si el resultado da un valor mayor a la precisión usada, los bits de la izquierda que sobren se truncan:

```
  0000 0101    (5)     (suma de dos positivos)
+ 0111 1000   (120)
= 0111 1101   (125)

  1111 1011    (-5)    (suma de un positivo y un negativo)
+ 0111 1000   (120)
= 0111 0011   (115)
```

En este último caso, el resultado completo era 1 0111 0011, pero el primer bit de la izquierda desborda los ocho bits de precisión usados, y por lo tanto se trunca. El resultado es 115, en binario...

```
   1111 1011     (-5)      (suma de dos negativos)
+  1111 1010     (-6)
=  1111 0101    (-11)
```

De nuevo, el resultado quedaría en 9 bits (*1 1111 0101*), pero el bit desbordado se trunca, quedando el valor (*-11*) en binario...

El hecho de poder hacer sumas y restas como si ambas fueran sumas, simplifica mucho los diseños de los circuitos sumadores de un computador. De hecho, en la práctica se requeriría sólo de dos circuitos: uno para complementar a dos y obtener números negativos (que sería simple...) y otro para sumar. Como el producto y la división son esencialmente sumas y restas, se puede ver que para hacer las cuatro operaciones básicas alcanzaría con sólo dos circuitos...

El problema final, es que si se suman dos positivos o dos negativos podría provocarse *desborde*: el resultado podría ser de valor absoluto tan grande que no sería posible de representar en la precisión usada. La consecuencia común de un desborde es que se obtiene un número de *signo cambiado* al que se esperaría:

```
   0000 0101     (5)       (suma de dos positivos, provocando desborde)
+  0111 1111   (127)
=  1000 0100   (-124)
```

El resultado debió dar *0 1000 0100* (o sea, 132) pero el bit de signo desbordó la precisión de ocho bits y *se truncó*, quedando un negativo que no se esperaba. Esta es la causa por la cual Java no permite asignar en forma directa un valor de un tipo mayor en una variable de un tipo menor: podría provocarse desborde en esa asignación. El programador debe usar casting explícito si quiere forzar la asignación, pero en ese caso, va por su propio riesgo!!! Por cierto, si esa suma se hace en precisión *short* (16 bits) el resultado cabe sin problemas.

Un detalle final que debe cuidar el programador, es que si una variable de un tipo menor se asigna en otra de tipo mayor, se mantiene el valor y el signo en la variable receptora (como era de esperarse...) Se dice que al asignar se produce *propagación de signo desde la variable menor hacia la mayor*. Esto es tan obvio que no parece valer la pena comentarlo: en el siguiente modelo, las variables a y b quedan valiendo 8...

```
byte a = 8;                      0000 1000
short  b = (short) a;      0000 0000 0000 1000
```

Si el valor de *a* es negativo, también se propaga el signo pero observe que el byte más significativo se llena con unos (y no con ceros...)

```
byte a = -8;                     1111 1000
short  b = (short) a;      1111 1111 1111 1000
```

3.) Operadores a nivel de bits en Java.

Java (como todo otro lenguaje) provee varios otros operadores para manejar bits. Mostramos una tabla con breve resumen de todos ellos:

Operador	Significado
&	*and* a nivel de bits
|	*or* a nivel de bits
^	*or exclusivo* a nivel de bits
~	*negación* a nivel de bits
<<	*desplazamiento a la izquierda* de bits, relleno con ceros a la derecha
>>>	*desplazamiento a la derecha* de bits, rellenando con ceros a la izquierda
>>	ídem anterior, pero *rellenando a la izquierda con el bit de signo*.

Todos ellos trabajan con los bits de los valores sobre los que se pide operar. Algunos ejemplos muestran la forma de usarlos (ver el modelo que acompaña a esta Ficha):

```
int n1 = 0x4C;    // 0100 1100
int n2 = 0x2F;    // 0010 1111
int n3 = n1 & n2;
int n4 = n1 | n2;
int n5 = n1 ^ n2;
int n6 = ~n1;
int n7 = n1 << 3;
int n8 = n6 >>> 3;
int n9 = n6 >> 3;

System.out.println("n1: " + n1);
System.out.println("n1 en binario: " + Integer.toBinaryString(n1));

System.out.println("\nn2: " + n2);
System.out.println("n2 en binario: " + Integer.toBinaryString(n2));

System.out.println("\nn1 & n2: " + n3);
 System.out.println("n1 & n2 en binario: " + Integer.toBinaryString(n3));

 System.out.println("\nn1 | n2: " + n4);
 System.out.println("n1 | n2 en binario: " + Integer.toBinaryString(n4));

 System.out.println("\nn1 ^ n2: " + n5);
 System.out.println("n1 ^ n2 en binario: " + Integer.toBinaryString(n5));

 System.out.println("\n~n1: " + n6 + " (n6)");
 System.out.println("~n1 en binario: " + Integer.toBinaryString(n6)+ " (n6)");

 System.out.println("\nn1 << 3: " + n7);
 System.out.println("n1 << 3 en binario: " + Integer.toBinaryString(n7));

 System.out.println("\nn6 >>> 3: " + n8);
 System.out.println("n6 >>> 3 en binario: " + Integer.toBinaryString(n8) + " (son 29 bits en pantalla...)" );
```

```
System.out.println("\nn6 >> 3: " + n9);
System.out.println("n6 >> 3 en binario: " + Integer.toBinaryString(n9)+ " son 32 bits en pantalla...)" );
```

Hay algunas aplicaciones muy útiles de estos operadores. Por ejemplo, podemos observar que el operador de desplazamiento de bits hacia la izquierda (<<) agrega un cero en la parte derecha de la secuencia por cada desplazamiento pedido:

```
short a = 2;            // 0000 0000 0000 0010  esto es un 2 en binario...
a = (short) (a << 1);   // 0000 0000 0000 0100  esto es un 4 en binario…
a = (short) (a << 1);   // 0000 0000 0000 1000  esto es un 8 en binario…
```

y puede verse fácilmente que cada desplazamiento de un bit a la derecha equivale a *multiplicar por dos* al valor de la variable... Este es un recurso muy usado, pues la implementación del desplazamiento es más eficiente y veloz que la multiplicación por medio del operador producto (*) Algo similar puede decirse del desplazamiento a la derecha: cada desplazamiento de un bit a la derecha, equivale a dividir por dos el valor.

El operador & puede ser muy valioso para capturar secuencias de bits y trasladarlas sin cambios a otras variables de más precisión, sin sufrir propagación de signo, o para aplicar otros trucos valiosos. Por ejemplo, suponga que se tiene una variable *x* asignada con alguna secuencia binaria, y se quiere saber cuánto vale el tercer bit menos significativo (o sea, el tercero empezando de la derecha). Podría parecer una tarea intimidante, pero puede hacerse fácilmente de la forma siguiente:

```
// sea x valiendo un valor cualquiera...
byte x = 0x4B;  //  en este caso: 0100 1011 y queremos consultar cuánto vale el  tercer bit

// usamos una máscara en la que sólo vale 1 el tercer bit...
byte m = 0x04;   // 0000 0100

// y hacemos un  and entre x y m...
byte r = (byte) (x & m);   // en este caso, quedará r = 0000 0000

// ahora el remate:
if( r == 0)
{
        // si entró por acá, el bit era cero... no hay otra forma en que el 1 de m se pueda
        // haber convertido en 0...
}
else
{
        // si entró por acá, el bit era uno... no hay otra forma en que el 1 de m se pueda
        // haber salvado de convertirse en 0...
}
```

Otra aplicación muy común, que involucra ahora al operador | (*or*), es la de asignar un *uno* en un bit específico (garantizar que un bit dado quede valiendo *uno*), sin cambiar ninguno de los otros. Para ello se puede aplicar el esquema siguiente:

```
// sea x valiendo un valor cualquiera...
```

short x = 0x004B; // en este caso: 0000 0000 0100 1011 y queremos en uno el quinto bit

// usamos una *máscara* en la que sólo vale 1 el quinto bit...
short m = 0x0010; // 0000 0000 0001 0000

// y hacemos un or entre x y m...
short r = (short) (x | m); // en este caso, quedará r = 0000 0000 0101 1011

4.) Caso de Análisis: Generación de Subconjuntos para un Conjunto de n Elementos.

Para mostrar la potencia general del manejo de bits, proponemos un conocido problema que suele aparecer en diversas ocasiones, ya sea por sí mismo o bien como parte o subproblema de otros problemas mayores: la *generación de los subconjuntos de un conjunto P de n elementos*.

Sea *P* un conjunto finito de *n* elementos cualesquiera. El valor *n* recibe el nombre de *cardinalidad* de *P*, y se denota también como $|P|$. El conjunto *PS* de todos los subconjuntos posibles de *P* (incluidos el conjunto vacío y el propio *P*) se designa como el *conjunto potencia de P*. En el siguiente ejemplo, mostramos un conjunto *P* con *n* = 3 elementos, y el conjunto *PS*, potencia de *P*, con todos sus elementos por enumeración:

P = {a, b, c} ⇨ |P| = n = 3
PS = {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}} ⇨ $|PS| = 2^n = 8$

Puede verse que el conjunto potencia *PS* contiene un elemento por cada subconjunto que hay en *P*, y que cada uno de esos subconjuntos no es otra cosa que una de las posibles *combinaciones* (formas distintas de combinar elementos, sin repeticiones) de los elementos de *P*.

Es útil notar que se puede establecer una *correspondencia uno a uno* (o sea, una *relación biyectiva*) entre los elementos del conjunto potencia *PS* de un conjunto *P*, y el conjunto *B* de secuencias *b* de *n* bits, en las que *b* tiene un *1* en cada posición (de derecha a izquierda) que corresponde a un elemento de *P* que esté presente en el subconjunto, o un *0* en caso contrario. Veamos esto más claro en el siguiente ejemplo:

Sea: P = {1, 2, 3} ⇨ |P| = n = 3
Entonces: PS = {{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

Por lo dicho, podemos representar a cada elemento en *PS* como una secuencia de *n = 3* bits, en la cual el primer bit de la derecha se use para indicar si el elemento *1* de *P* está en ese subconjunto (*bit = 1*) o no (*bit = 0*). El *subconjunto vacío {}* no contiene a ninguno de los tres elementos de *P*, por lo que se representa con una secuencia de tres ceros: *000* El subconjunto *{1}* solo contiene al *1* de *P*, y se representa con la secuencia *001*, y así sucesivamente. La tabla siguiente muestra la correspondencia completa, indicando además el valor numérico en base 10 (o *valor nominal*) que le corresponde a cada secuencia de bits así obtenida:

Subconjunto	**Secuencia de n=3 bits**	**Valor numérico (en base 10)**
{}	000	0
{1}	001	1
{2}	010	2
{3}	100	4

{1,2}	011	3
{1,3}	101	5
{2,3}	110	6
{1,2,3}	111	7

Si cada elemento de *PS* se corresponde biyectivamente con un valor binario de *n* bits, y sabiendo que con *n* bits podemos formar 2^n combinaciones diferentes, se deduce que el conjunto potencia *PS* tendrá exactamente 2^n elementos, siendo *n* la cardinalidad de *P*: $|PS| = 2^n$.

Lo anterior muestra que si queremos plantear un algoritmo que genere todos los subconjuntos de un conjunto *P* de *n* elementos, no es necesario (ni útil…) crear listas o arreglos individuales por cada subconjunto: podemos *mapear* a cada subconjunto en cada uno de los 2^n números del intervalo representado por $[0 .. (2^n) - 1]$.

Suponga entonces que se quiere generar una lista *S* con todos los subconjuntos de *P*, siendo *P* el conjunto *{1, 2, 3, …, n}*. Un esquema de pseudocódigo para hacerlo sería el siguiente:

subsets(n):

- Sea t = 2^n la cantidad de subconjuntos a generar.
- Sea S una lista vacía.
- Para x = 0 hasta x = t – 1 repetir:
  - Agregar x en S
- Retornar S

El algoritmo anterior produce una lista *S* en la que cada elemento es un número entero que representa a cada una de las posibles combinaciones de *n* elementos. La cuestión ahora es que en la práctica, en algún momento se tendrá uno de estos subconjuntos *s*, y se querrá saber si el valor *x* pertenece a *s* o no. Sea el conjunto *s = {2,3} = 6 = 110* de la tabla anterior. Suponga que queremos chequear si x = 2 pertenece a *s* (visualmente podemos confirmar que sí, ya que en la secuencia 110 el primer 1 desde la derecha corresponde a la posición del 2) Se pueden usar operadores de bit para hacer la comprobación: concretamente, la instrucción

r = s & (1 << (x-1))

dejará en *r* el resultado de esa comprobación: si *r = 0*, entonces *x* NO pertenecía a *s*. La expresión *1 << (x-1)* toma el byte con el valor *1* a la derecha, y lo desplaza *x-1* bits a la izquierda. Si *x = 2*, entonces:

001 << (2-1) ⇨ 001 << 1 ⇨ 010

Se puede ver que el resultado es la secuencia *010*, que solo contiene un *1* y en la posición que correspondería a *x = 2* en el conjunto (lo cual a su vez, equivale al conjunto *{2}*) La parte final de la instrucción hace un *and* entre el conjunto *s* y el valor *010 = {2}* recién obtenido:

r = s & 010 = 110 & 010 = 010

El resultado final en *r* es *010*, que solo podría haberse obtenido si *s* contenía un *1* en el segundo bit. Esto muestra que si *r != 0* entonces *s* contenía a *x*, pero si *r == 0* no lo contenía.

En forma similar, podemos comprobar que las expresiones siguientes, agregan *x* a un conjunto *s* o remueven *x* de *s*, respectivamente:

```
s = s | ( 1 << (x-1) )      // agrega el valor x al conjunto s
s = s & ~( 1 << (x-1) )     // remueve el valor x de s
```

En algunas ocasiones el problema de la generación de subconjuntos aparece con ligeras variantes (que lo hacen aun más interesante). Por ejemplo, en algunas implementaciones del *Problema del Viajante* basadas en *programación dinámica*, se supone un conjunto de *n* ciudades numeradas entre *1* y *n*, conectadas todas con todas en forma directa (se supone que existe un arco entre cualquier par de ciudades que se elija). En una de las fases del problema, se requiere contar con una lista que contenga todas las posibles formas de pasar desde la ciudad *1* hasta la ciudad *k*, *(con 1<= k <= n)*, pero pasando solo por ciudades numeradas entre *1* y *k*. A modo de ejemplo, suponga *n = 4* ciudades con lo que *P = {1, 2, 3, 4}* es el conjunto de ciudades. Las combinaciones de caminos pedidas son:

k = 1
S_1 = {combinaciones que comienzan en 1 y llegan a k=1, pasando por 1}
S_1 = {{1}}

k = 2
S_2 = {combinaciones que comienzan en 1 y llegan a k=2, pasando por 1 y 2}
S_2 = {{1,2}}

k = 3
S_2 = {combinaciones que comienzan en 1 y llegan a k=3, pasando por 1, 2 y 3}
S_2 = {{1,3}, {1,2,3}}

k = 4
S_2 = {combinaciones que comienzan en 1 y llegan a k=4, pasando por 1, 2, 3 y 4}
S_2 = {{1,4}, {1,2,4}, {1,3,4}, {1,2,3,4}}

¿Cómo generar la lista de subconjuntos en este caso? Note que todos los subconjuntos contienen al *1*, y que no se permiten todas las combinaciones posibles, sino solo aquellas que se ajustan a salir de la ciudad *1* y pasar por ciudades numeradas entre *1* y *k*, con *k = 1, 2, 3, … n.* La idea es simple: calcular el conjunto potencia de *P – {1} = {2,3,4,…,n}*. Esto generará todas las combinaciones de todos contra todos sin incluir al 1, y como ahora hay *n-1* elementos, el conjunto potencia *PS(P-{1})* tendrá $t = 2^{(n-1)}$ elementos. Considerando el mismo ejemplo anterior de *n = 4* ciudades, quedaría algo como:

P = {1,2,3,4} [n = 4] ⇨ P – {1} = {2,3,4} [t = $2^{(n-1)} = 2^3 = 8$]
PS(P–{1}) = {{}, {2}, {3}, {4}, {2,3}, {2,4}, {3,4}, {2,3,4}}

Y ahora, a cada subconjunto de *PS* simplemente añadirle el *1*, lo cual puede hacerse con sencillez: si s es uno de los elementos PS (por ejemplo, *{2} = 0010*) entonces la operación de agregar un *1* a *s* puede plantearse como: *s = (s << 1) + 1*:

PS+{1} = {{1}, {1,2}, {1,3}, {1,4}, {1,2,3}, {1,2,4}, {1,3,4}, {1,2,3,4}}

Obviamente, estos subconjuntos también se pueden representar como secuencias de *n* bits: en este caso, las secuencias son *{0001, 0011, 0101, 1001, 0111, 1011, 1101, 1111}* que corresponden a los valores nominales *{1, 3, 5, 9, 7, 11, 13, 15}*.

Proponemos un algoritmo sencillo en pseudocódigo, para hacer la generación de todos los subconjuntos en las condiciones dadas:

subsets2(n):

- Sea n = la cantidad de ciudades (vértices de la red).
- Sea t = $2^{(n-1)}$ la cantidad de subconjuntos a generar.
- Sea S una lista vacía.
- Para x = 0 hasta x = t – 1 repetir:
  - Sea sx = (x << 1) + 1
  - Agregar sx en S
- Retornar S

Los detalles de implementación, y su aplicación concreta para un problema puntual, se dejan para el alumno.