Binary Numbers and Bit Manipulation

Authored by AllisonP

Table Of Contents

Radix (Base)

The number of digits that can be used to represent a number in a positional number system. The decimal number system (base-10) has 10 digits ($0, 1, 2, 3, 4, 5, 6, 7, 8, 9$); the binary (base-$2$) number system has 2 digits ($0, 1$).

We think in terms of base-10, because the decimal number system is the only one many people need in everyday life. For situations where there is a need to specify a number's radix, number n having radix r should be written as $(n)_r$.

Binary to Decimal Conversion

In the same way that $(840)_{10} = (8 \times 10^2) + (4 \times 10^1) + (0 \times 10^0) = 800 + 40 + 0 = 840$, a binary number having k digits in the form of $d_{k-1}d_{k-2} \dots d_2 d_1 d_0$ can be converted to decimal by summing the result for each $d_i \times 2_i$ where $0 \le i \le k-1$, $i = k-1$ is the most significant bit, and $i = 0$ is the least significant bit.

For example: $(1011)_2 \to (?)_{10}$ is evaluated as
$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = 8 + 0 + 2 + 1 = (11)_{10}$

Decimal to Binary Conversion

To convert an integer from decimal to binary, repeatedly divide your base-10 number, n, by 2. The dividend at each step i should be the result of the integer division at each step $i - 1$. The remainder at each step of division is a single digit of the binary equivalent of n; if you then read each remainder in order from the last remainder to the first (demonstrated below), you have the entire binary number.

For example: $(4)_{10} \to (?)_2$. After performing the steps outlined in the above paragraph, the remainders form $(100)_2$ (the binary equivalent of $(4)_{10}$) when read from the bottom up:

$$4 \div 2 = 2 \text{ remainder } 0 \uparrow$$
$$2 \div 2 = 1 \text{ remainder } 0 \uparrow$$
$$1 \div 2 = 0 \text{ remainder } 1 \uparrow$$

This can be expressed in pseudocode as:

```
while(n > 0):
    remainder = n % 2;
    n = n / 2;
    Insert remainder to front of a list or push onto a stack

Print list or stack
```

Go to Top

Many languages have built-in functions for converting numbers from decimal to binary. To convert an integer, n, from decimal to a String of binary numbers in Java, you can use the *Integer.toBinaryString(n)* function.

Note: The algorithm discussed here is for converting integers; converting fractional numbers is a similar (but different) process.

Logical Statements and Boolean Algebra

If you're not familiar with the term *boolean algebra*, you might be surprised to know that you likely already learned how to do this during the *logic* unit commonly taught during High School (secondary education). If you need a refresher, check out the Wikipedia article on Truth Tables.

Instead of using T (true) and F (false), boolean algebra uses the binary numbers 1 for *true* and 0 for *false*.

Basic Operators

Here are some commonly used Java operators you should familiarize yourself with:

- **&** *Bitwise AND* ($\wedge$). This binary operation evaluates to 1 (true) if both operands are true, otherwise 0 (false). In other words:

```
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

- **|** *Bitwise Inclusive OR* ($\vee$). This binary operation evaluates to 1 if *either* operand is true, otherwise 0 (false) if both operands are false. In other words:

```
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

- **^** *Bitwise Exclusive OR* or *XOR* ($\oplus$). This binary operation evaluates to 1 (true) if and only if exactly one of the two operands is 1; if both operands are 1 or 0, it evaluates to 0 (false). In other words:

```
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

- **~** The unary *Bitwise Complement* operator flips every bit; for example, the bitwise-inverted 8-bit binary number 01111001 becomes 10000110, and the bitwise-inverted signed decimal integer 8 becomes -9.

Example: Converting to Binary

- EXAMPLE

The Java code below converts a single string of characters to a binary string:

Go to Top

```
import java.util.*;

class BinaryString {

    BinaryString(String string){
        for( byte b : string.getBytes() ){
            System.out.print(Integer.toBinaryString(b) + " ");
        }
        System.out.println();
    }

    BinaryString(Integer integer){
        System.out.println(Integer.toBinaryString(integer));
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        new BinaryString(scanner.next());
        scanner.close();
    }
}
```

Input

```
HackerRank
```

Run

Output

Solution

When run, this code prints the following output:

```
1001000 1100001 1100011 1101011 1100101 1110010 1010010 1100001 110111
0 1101011
```

which is binary for `H a c k e r R a n k`.

- EXAMPLE

Let's modify our Java BinaryString class to find and print the *OR* of each character in `HackerRank` with 8675309:

```
import java.util.*;

class BinaryString {

    BinaryString(String string, Integer integer){
        String binaryInteger = Integer.toBinaryString(integer);

        for( byte b : string.getBytes() ){
            // Perform a bitwise operation using byte and integer op
            int tmp = b | integer;
            System.out.println( Integer.toBinaryString(b) + " OR " +
                + " = " + Integer.toBinaryString(tmp) + " = " + tmp
        }
    }

    public static void main(String[] args) {
```

Go to Top

```
        Scanner scanner = new Scanner(System.in);
        String s = scanner.next();
        Integer i = scanner.nextInt();
        new BinaryString(s, i);
        scanner.close();
    }
}
```

Input

```
HackerRank 8675309
```

Run

Output

Solution

The above code produces the following output:

```
1001000 OR 100001000101111111101101 = 100001000101111111101101 = 86753
09
1100001 OR 100001000101111111101101 = 100001000101111111101101 = 86753
09
1100011 OR 100001000101111111101101 = 100001000101111111101111 = 86753
11
1101011 OR 100001000101111111101101 = 100001000101111111101111 = 86753
11
1100101 OR 100001000101111111101101 = 100001000101111111101101 = 86753
09
1110010 OR 100001000101111111101101 = 100001000101111111111111 = 86753
27
1010010 OR 100001000101111111101101 = 100001000101111111111111 = 86753
27
1100001 OR 100001000101111111101101 = 100001000101111111101101 = 86753
09
1101110 OR 100001000101111111101101 = 100001000101111111101111 = 86753
11
1101011 OR 100001000101111111101101 = 100001000101111111101111 = 86753
11
```

Notice that the first 17 bits (10000100010111111) are always the same. This is because bit position is counted starting with the least-significant (rightmost) bit and then it moves left so, in the example above, the only values with the *potential* to change are the lower (rightmost) 7 bits (as that is the number of bits in the smaller operand). For each bit position in the lower 7 bits, an *OR* operation is performed. If we were to again modify the above code to print the *exclusive OR* (instead of the inclusive OR), we would get this output:

Go to Top

```
1001000 XOR 1000010001011111111101101 = 1000010001011111110100101 = 8675
237
1100001 XOR 1000010001011111111101101 = 1000010001011111110001100 = 8675
212
1100011 XOR 1000010001011111111101101 = 1000010001011111110001110 = 8675
214
1101011 XOR 1000010001011111111101101 = 1000010001011111110000110 = 8675
206
1100101 XOR 1000010001011111111101101 = 1000010001011111110001000 = 8675
208
1110010 XOR 1000010001011111111101101 = 1000010001011111110011111 = 8675
231
1010010 XOR 1000010001011111111101101 = 1000010001011111110111111 = 8675
263
1100001 XOR 1000010001011111111101101 = 1000010001011111110001100 = 8675
212
1101110 XOR 1000010001011111111101101 = 1000010001011111110000011 = 8675
203
1101011 XOR 1000010001011111111101101 = 1000010001011111110000110 = 8675
206
```

If you're still having some trouble understanding how bitwise operations work, spend some time comparing the different outputs and experimenting with the code that produced them.

Additional Resources

Java: Summary of Operators