Bit Manipulation

Micah Stairs

Outline

- Overview
- Review of Binary Numbers
- Integers and Longs
- Bitwise Operators
- Bit Operations
- Sets of Elements
- BitSet Class
- Solution Sketch for a Kattis Problem

Overview

Bit manipulation is a fundamental concept in Computer Science. In competitive programming, bit manipulation has a number of applications:

- Representing information in a compact way
- Performing fast computations
- Representing sets of elements

Review of Binary Numbers

As opposed to representing a number in the typical base-10 numeral system (with digits 0-9), the base-2 (or binary) numeral system is represented using only the digits 0 and 1.

Each digit is referred to as a "bit". The most significant bits are on the left and the least significant bits are on the right.

$$10010011_2 = 147_{10}$$

$$(1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)$$

Integers and Longs

Integers and Longs

In Java, an *int* is 32 bits and a *long* is 64 bits.

They are both ‘signed’, which means they have a range of both positive and negative values that can be represented.

Integers and Longs

The left-most bit is used to indicate whether or not the number is negative, leaving the remaining 31 or 63 bits to represent the value.

An *int* can represent a number in the range $[-2^{31}, 2^{31}-1]$.

A *long* can represent a number in the range $[-2^{63}, 2^{63}-1]$.

Integers and Longs

If any of your computations exceed the range of your data type, then you will become a victim of integer overflow. This will give you unexpected (wrong) answers.

If in doubt, pick the larger data type. If *long* isn't large enough, you may need to resort to the *BigInteger* class.

Integers and Longs

When initializing *int* (or *long*) variables, there is more than one type of notation that can be used. The following are equivalent:

```
int val1 = 1243921300;
int val2 = 1_243_921_300;
int val3 = 0b1001010001001001011101110010100;
int val4 = 0b_001001010_00100100_10111011_10010100;
```

The underscores can be added to improve readability. The "0b" is used to indicate the number will be specified in binary.

Integers and Longs

When initializing *long* variables which are outside the bounds of what can fit inside an *int*, the number must be followed by a lowercase or uppercase "L" (preferably the latter).

```
long MOD = 1_000_000_000_007L;
```

Forgetting to do so will cause a compilation error.

Integers and Longs

When possible, it is best to use the primitive *int* and *long* types as opposed to Integer and Long objects. The primitive types use less space and are faster.

The main reason the object wrapper is useful is because of generics. Primitive types cannot be used directly inside generic classes since a reference is required.

Invalid:

```
List<int> list = new ArrayList<>();
```

Valid:

```
List<Integer> list = new ArrayList<>();
```

Valid:

```
List<int[]> list = new ArrayList<>();
```

Integers and Longs

Another reason to use the objects would be if you need to use the “null” value.

The Integer and Long classes, however, contain a number of static methods which take and return primitive types.

Integers and Longs

Method	Description
toBinaryString(value)	formats number as binary string **Ex:** 37 -> “100101”
bitCount(value)	counts bits
reverse(value)	reverses the order of the bits
rotateLeft(value, distance)	rotates bits (wrapping around)
rotateRight(value, distance)	rotates bits (wrapping around)

Integers and Longs

Method	Description
numberOfLeadingZeroes(value)	counts leading zeroes in bit pattern
numberOfTrailingZeroes(value)	counts trailing zeroes in bit pattern
highestOneBit(value)	clears all bits except highest set bit
lowestOneBit(value)	clears all bits except lowest set bit

Bitwise Operators

Bitwise Operators

Symbol	Operation
~	NOT (negates bit pattern)
&	AND
|	inclusive OR
^	exclusive OR
<<	signed left shift
>>	signed right shift
>>>	unsigned right shift

Unary and Binary Bitwise Operators

NOT

```
        1 0 1 0 1 0
-----------------------
1 ... 1 0 1 0 1 0 1 0 1
```

AND

```
      1 1 0 1 0
  1 0 1 0 1 0
-------------
      1 0 1 0
```

OR

```
      1 1 0 1 0
  1 0 1 0 1 0
-------------
  1 1 1 0 1 0
```

XOR

```
      1 1 0 1 0
  1 0 1 0 1 0
-------------
  1 1 0 0 0 0
```

Bit Shifting to the Left

```
int val = 0b1011;
int shifted = val << 2;
System.out.println(Integer.toBinaryString(shifted));
```

Outputs: “101100”

```
int val = 0b1011;
int shifted = val << 31;
System.out.println(Integer.toBinaryString(shifted));
```

Outputs: “10000000000000000000000000000000”

Note: Shifting *int* values by more than 31 or *long* values by more than 63 has undefined behaviour.

Bit Shifting to the Right

```
int val = 0b1011;
int shifted = val >> 1;
System.out.println(Integer.toBinaryString(shifted));
```

Outputs:“101”

```
int val = 0b1011;
int shifted = val >>> 2;
System.out.println(Integer.toBinaryString(shifted));
```

Outputs: “10”

Note: The signed and unsigned right bit shifting behave the same when working with positive values.

Bit Shifting to the Right

```
System.out.println(Integer.toBinaryString(-1));
```

Outputs “11111111111111111111111111111111”

```
System.out.println(Integer.toBinaryString(-1 >> 16));
```

Outputs “11111111111111111111111111111111”

```
System.out.println(Integer.toBinaryString(-1 >>> 16));
```

Outputs “1111111111111111”

Bit Operations

Basic Bit Operations

These are some of the most common operations that you will need to do when dealing with a set of bits.

Note: $2^n = 1 << n$

```
int setBit(int set, int i) {
  return set | (1 << i);
}

boolean isSet(int set, int i) {
  return (set & (1 << i)) != 0;
}

int clearBit(int set, int i) {
  return set & ~(1 << i);
}

int toggleBit(int set, int i) {
  return set ^ (1 << i);
}

int setAll(int n) {
  return (1 << n) - 1;
}
```

Advanced Bit Operations

But there are lots of other cool things that can be done.

```
boolean isEven(int val) {
  return (val & 1) == 0;
}

boolean isPowerOfTwo(int n) {
  return n > 0 && (n & (n - 1)) == 0;
}

boolean oppositeSigns(int a, int b) {
  return (a ^ b) < 0;
}

int modByPowerOfTwo(int val, int power) {
  return val & ((1 << power) - 1);
}
```

How is this typically done without bit manipulation?

Advanced Bit Operations

You can even swap two variables without creating a new temporary variable.

```
int a = 7;
int b = 13;
a ^= b;
b ^= a;
a ^= b;
System.out.println(a);
System.out.println(b);
```

Outputs “13” and then “7”

Sets of Elements

Bitset

In a set, each element appears either 0 or 1 times. This lends itself quite nicely to a bitstring representation.

An *int* can represent up to 32 elements and a *long* can represent up to 64 elements.

Each element will be assigned to a particular bit in the number.

Combinations

Using bit manipulation, it is quite easy to iterate over all possible combinations of some number of elements. This is known as the power set.

Note: Looping over **[0, 1 << n)** is the same as looping over all binary strings of length *n*.

```
int nElements = 20;
int nCombinations = 1 << nElements;
for (int c = 0; c < nCombinations; c++) {

  // Do something with the combination 'c'

}
```

Complement

Given some subset, you can also find the complement (the missing elements).

```
int nElements = 5;
int nCombinations = 1 << nElements;

int combination = 0b01100;

int complement = (nCombinations - 1) - combination;
System.out.println(Integer.toBinaryString(complement));
```

Outputs “10011”

BitSet Class

BitSet Class

If you need more than 64 bits you can either use multiple *int* / *long* values, or you can take advantage of the BitSet class (arbitrary number of bits).

Internally, this class stores the bits in a series of *long* values (although you do not have direct access to these values).

This class has all of the methods that you might expect.

Solution Sketch

Geppetto

https://open.kattis.com/problems/geppetto

Summary: Given a number of ingredients (up to 20) and a list of bad pairs of ingredients (up to 400), count the number of valid pizza combinations.

Geppetto

Solution:

- Read in the input, storing each bad pair of ingredients.
- Since there are only 20 ingredients we can iterate over all combinations. For each combination, we loop over all bad pairs of ingredients. A combination is counted if it doesn't contain any of these bad pairs of ingredients (checked using bit manipulation).

Sources

- https://docs.oracle.com
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html
- http://docs.oracle.com/javase/7/docs/technotes/guides/language/binary-literals.html
- https://open.kattis.com/problems/geppetto