**Using Two Longs to Encode the Board**

In Java and in many other programming languages, a so-called long integer is composed of eight bytes. Eight bytes are 64 bits (1 byte = 8 bits). If presented as a series of ones and zeros, the most significant bit (MSB) is on the left hand side, the least significant bit (LSB) is on the right hand side. That's why we number the bits in the way shown below from right to left. Remember, programmers start counting with zero ;-)

63 62 61 60 ... 48 47 46 45 ... 3 2 1 0

Connect Four is played on a vertical board with seven columns and six rows. That makes 42 slots. The board with these 42 slots is shown in the diagram below. We add an additional row on top for convenience purposes. This additional row on top is for computational reasons only. And so are the bits numbered 49 to 63, adding two more columns and a bit. The bits of the top row (6, 13, 20, etc.) and the bits on the right (49 - 63) are seemingly unused but nonetheless important and not to forget in their role when it comes to manipulating the bits.

The numbers indicate the position in the bit representation of a long integer.

6 13 20 27 34 41 48 55 62 Additional row

+---------------------+

| 5 12 19 26 33 40 47 | 54 61 top row

| 4 11 18 25 32 39 46 | 53 60

| 3 10 17 24 31 38 45 | 52 59

| 2 9 16 23 30 37 44 | 51 58

| 1 8 15 22 29 36 43 | 50 57

| 0 7 14 21 28 35 42 | 49 56 63 bottom row

+---------------------+

Let's take an example, given the following position:

. . . . . . .

. . . . . . .

. . . . . . .

. . . O . . .

. . . X X . .

. . O X O . .

-------------

0 1 2 3 4 5 6 colum (counting starts with 0 from left to right)

Connect Four is played by two players. There is one bitboard for each player. We encode the position separately for each player. One bitboard encodes the position with tokens X only, the other bitboard the position with tokens O only.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 13 20 27 34 41 48

. . . . . . . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 12 19 26 33 40 47

. . . . . . . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 11 18 25 32 39 46

. . . . . . . 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 10 17 24 31 38 45

. . . O . . . 0 0 0 0 0 0 0 0 0 0 1 0 0 0 2 9 16 23 30 37 44

. . . X X . . 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 8 15 22 29 36 43

. . O X O . . 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 7 14 21 28 35 42

-------------

0 1 2 3 4 5 6

Shown as a flat series of ones and zeros the two bitboards look like this. Leading zeros, i.e. the bits beyond position 48, are cut off.

... 0000000 0000000 0000010 0000011 0000000 0000000 0000000 // encoding Xs

... 0000000 0000000 0000001 0000100 0000001 0000000 0000000 // encoding Os

col 6 col 5 col 4 col 3 col 2 col 1 col 0

Got it? It's not that hard to understand, is it? This encoding is the key to the magic that makes bitboards so fast.

Subsequently, we assume that both bitboards are stored in an array declared long[] bitboard. The array contains only two bitboards (two long integers), of course: one encoding the Xs on the board, the other one encoding the Os.

**Bitwise Operations**

To work with a series of bits you need operations to manipulate them. All we need is two kinds of operations: to shift bits and to combine bits.

**Shifting Bits**

The operators >> and << shift bits to the right and to the left, respectively.

Saying e.g. 0b10101110 >> 3 means that three bits are shifted out to the right with three zeros being shifted in on the left. Here, the prefix 0b indicates that the following digits represent a binary number and not a decimal number. Assumed that 0b10101110 is a binary number of maximum eight bits, the result is 0b00010101.

Accordingly, 0b10101110 << 3 shifts three bits out to the left and shifts in three zeros on the right. The result is 0b01110000 if we assume a length of eight bits.

**Combining Bits**

The only two operators we need in this context are the so-called XOR-operator and the AND-operator.

The XOR-operator (written as ^) takes two binary numbers, say 0b10101110 ^ 0b10011111, and compares bits of the same position. If both bits are not equal, the result is a one; otherwise it's a zero. That's the rule for the XOR-operator.

If we write the second number below the first number, the bit positions are easier to compare. Only pairs of 10 and 01result in a 1.

10101110

^ 10011111

----------

00110001

That is, 0b10101110 ^ 0b10011111 results in 0b00110001.

The AND-operator (written as &) also takes two binary numbers. Comparing the bits of both numbers, it returns a one only if the first bit *and* the second bit are one; therwise it returns zero for the position under consideration. For example

10101110

& 10011111

----------

10001110

In other words, 0b10101110 & 0b10011111 results in 0b10001110.

For more information on bit shifting and other bit operations such as OR, AND and NOT you might consult Wikipedia on [Bitwise Operations](https://en.wikipedia.org/wiki/Bitwise_operation).

**Let's Play with the Bitboards**

To demonstrate the use of bitboards we implement four methods: make a move (makeMove), undo a move (undoMove), check whether there are four in a row (isWin), and list all possible moves in a given situation (listMoves). As our means of communication, we use pseudo-code which is close to TypeScript and Java. Only slight adaptations are required for any of the two languages. It is up to you to properly embed this code in a class statement.

Before we look into the methods, we need some few data the methods operate on: height, counter and moves. Together with the bitboard array (see above) the implementation of the methods is almost trivial. So, pay a little bit attention to the data required.

**Remember the Fill Level**

Before we look into the methods more closely, there is an additional piece of information tracked that speeds up making and undoing moves. The position to be filled next in a column is remembered in an array declared int[] height. For the example shown,

6 13 20 27 34 41 48

. . . . . . . 5 12 19 26 33 40 47

. . . . . . . 4 11 18 25 32 39 46

. . . . . . . 3 10 17 24 31 38 45

. . . O . . . 2 9 16 23 30 37 44

. . . X X . . 1 8 15 22 29 36 43

. . O X O . . 0 7 14 21 28 35 42

-------------

0 1 2 3 4 5 6

the values for height are:

{0, 7, 15, 24, 30, 35, 42}

The variable height serves as a memory where the next token goes given the column. Otherwise we would need to search for the next empty slot given the bottom of a column. Having a memory saves searching and makes things faster. And that is what we are up to with bitboards. We want to be as fast as possible with manipulating a board representation of Connect Four.

**Have a Counter**

Another piece of information we track is the number of moves done, a counter. Each time we make a move we increment the counter, each time we undo a move we decrement the counter. For incrementation we use the increment operator ++(add one), for decrementation we use the decrement operator -- (subtract one). We use incrementation and decrementation also for manipulating the height per column.

There is a neat trick to determine whether it is X's or O's turn to move given the counter. If counter is even, it's X's turn, if it's odd, it's O's turn (you might do it the other way around if you like). If you think about counter in its binary form, counter is even if the rightmost bit (the LSB) is zero and it is odd if the LSB is one. Using the AND-operator, counter & 1 returns 0 for even counts and 1 for odd counts. We will use this trick to determine, which bitboard of the two we are interested in.

**Maintain a History of Moves Done**

In a variable called moves, actually an array declared int[] moves, we remember the moves done so far. The moves are represented by the columns given when a move is made. moves[0] stores the first move, moves[1] the second and so on. We use counter to write to the right cell in moves when we make a move as well as to recall the last move, when we undo it.

**Make a Move**

Have a look at the code that describes how to make a move given a column; the column being an integer value ranging from 0 to 6 including both limiting values. It's just three line of code in the body of makeMove.

void makeMove(int col) {

long move = 1L << height[col]++; // (1)

bitboard[counter & 1] ^= move; // (2)

moves[counter++] = col; // (3)

}

The code says:

1. Given the column col, get the index(!) of the position stored in height for that column, shift a single bit (1L) to that position in the binary representation of the bitboard and store the result in move; afterwards (because the increment operator is in postfix position), height[col] is incremented by one.
2. bitboard[counter & 1] gets us the bitboard of the party (either X or O) on turn. The bit move is simply set via the XOR-operator on the corresponding bitboard. bitboard[counter & 1] ^= move; is a shortcut for bitboard[counter & 1] = bitboard[counter & 1] ^ move;.
3. Store the column col in the history of moves, afterwards (because ++ is again in postfix position) increment the counter.

Even if you are not into the details of a programming language (which makes it three lines of code instead of, say, five), you should get the idea of what's going on.

**Undo a Move**

Method undoMove basically reverses the steps done in makeMove. See yourself:

void undoMove() {

int col = moves[--counter]; // reverses (3)

long move = 1L << --height[col]; // reverses (1)

bitboard[counter & 1] ^= move; // reverses (2)

}

1. The column (col) is determined from the history of moves; prior to accessing the history (because -- is in prefix position), counter is decremented. This step "reverses" the last action in makeMove, so to speak.
2. We decrement and store the new height for col first (again, -- is in prefix position), take the new value and shift a bit (1L) to the corresponding position in the bit pattern and store it in move.
3. We unset the bit in the corresponding bitboard. This step "reserves" the second action in makeMove, the setting of the bit.

**Are there Four In A Row?**

We are getting to the heart, why we are using bitboards. Since we need to check for four in a row almost every time a move is made, it is in our interest to get the check done as quickly as possible. Speed improves the number of positions investigated per second, and the sheer number of positions crunched correlates with the perceived computer's "intelligence" of play.

Look again at the encoding of positions in a bitboard.

6 13 20 27 34 41 48 55 62 Additional row

+---------------------+

| 5 12 19 26 33 40 47 | 54 61 top row

| 4 11 18 25 32 39 46 | 53 60

| 3 10 17 24 31 38 45 | 52 59

| 2 9 16 23 30 37 44 | 51 58

| 1 8 15 22 29 36 43 | 50 57

| 0 7 14 21 28 35 42 | 49 56 63 bottom row

+---------------------+

To check if there are four in a row horizontally, we need for example to look at positions 11, 18, 25 and 32 whether they are all set to one. Likewise, we might check positions 21, 28, 35 and 42. The underlying positional pattern is that the positions looked at differ by a constant number of 7. Take the leftmost position (here 11 and 21, respectively), add 7 and 7 and 7 again, you get the four positions to look at.

Vertically, the difference is 1, take position 15 and 16 as an example. Diagonally it's either a difference of 8 (take 16 and 24 as an example) or a difference of 6 (take 30 and 36).

So, the "magic" difference numbers on the bitboard are 1, 6, 7 and 8.

Let's take a bitboard and shift a copy of it by 6 to the right, another copy by twice as much, and a final copy by thrice as much to the right. Then let's "overlay" all copies with the AND-operator. The effect is that all of the vertically distributed positions of the bitboard making up four in a row (vertically) are now queried: Is each one of you a one? If so, we identified four bits in a row.

The point is that bit shifting and combining bits make it a parallel computation for all positions on the board! You don't look at an individual position, you look at all the positions on the board at once and check for their neighbors being another three bits set as well.

If you get the basic idea, the following code should come as no surprise. The code literally does what we just described.

boolean isWin(long bitboard) {

if (bitboard & (bitboard >> 6) & (bitboard >> 12) & (bitboard >> 18) != 0) return true; // diagonal \

if (bitboard & (bitboard >> 8) & (bitboard >> 16) & (bitboard >> 24) != 0) return true; // diagonal /

if (bitboard & (bitboard >> 7) & (bitboard >> 14) & (bitboard >> 21) != 0) return true; // horizontal

if (bitboard & (bitboard >> 1) & (bitboard >> 2) & (bitboard >> 3) != 0) return true; // vertical

return false;

}

Notice that the additional row on the top of the regular board (positions 6, 13, 20, 27, 34, 41, 48) and the additional rows at the right (position 49 and up) are essential. They hold values of zero and are vital to prevent us from finding four in a row crossing "boarders". For example, checking the diagonal 9, 15, 21 has 27 next; "luckily", position 27 -- being a member of the additional top row -- is always zero. Without these additional row and columns our bitboards would not work.

The code above is highly regular and redundant. We might choose to capture the "magic" numbers in an array called directions and iterate over the numbers in a for-loop.

boolean isWin(long bitboard) {

int[] directions = {1, 7, 6, 8};

for(int direction : directions)

if (bitboard & (bitboard >> direction) &

(bitboard >> (2 \* direction)) & (bitboard >> (3 \* direction)) != 0)

return true;

return false;

}

You might catch the point that shifting a bitpattern once, twice and thrice is a bit of overdoing things. In case you want to save one shift operation and a multiplication by 3, you can use an intermediate variable called bb and get a little bit better off. In terms of machine code this saves you a few cycles of computations. Other than that the following code does exactly the same as our more verbose version of isWin does.

boolean isWin(long bitboard) {

int[] directions = {1, 7, 6, 8};

long bb;

for(int direction : directions) {

bb = bitboard & (bitboard >> direction);

if (bb & (bb >> (2 \* direction)) != 0) return true;

}

return false;

}

Per direction the routine needs one assignment, two shifts, two binary AND-operations, one multiplication, and a comparison, i.e. seven operations which relate to almost seven machine code or byte code instructions. A full evaluation considering all four directions requires four times that much, that is 4 x 7 = 28 instructions. Including the instructions required for the for-statement, we might end up with 30-something instructions altogether.

On a machine crunching 2 billion instructions per second, we can run 50 million checks of isWin per second (assuming 40 instructions per check). That's quite a number.

**Generate Valid Moves**

Last but not least, we need a generator to identify possible moves in a given situation of play. The list of moves (int[] moves) is calculated by listMoves and returned, each value representing a column which is not full yet.

int[] listMoves() {

int[] moves;

long TOP = 1000000\_1000000\_1000000\_1000000\_1000000\_1000000\_1000000L;

for(int col = 0; col <= 6; col++) {

if (TOP & (1L << height[col]) == 0) moves.push(col);

}

return moves;

}

We do so with a helper variable called TOP, which encodes the positions of the additional row on top of the board (positions 6, 13, 20, 27, 34, 41, 48). When we simulate the code for making a move (shifting a bit by height[col]), that bit is not supposed to land in the additional top row, marking the column full. If the column is not full, the column gets listed in moves.