# Bit manipulation

1's complement | 2's complement | and | bits | bitwise operations | or | xor

---

Original Article:

- A Bit of Fun: Fun with Bits (by bmerry)

## Number representation

### 1's complement

Given an 8 bit binary number `00010110`, its 1's complement is given by just flipping all its bits i.e. `11101001`.
The sign of the number can be indentified using the MSB and in 1's complement `0` has two representations `+0` as `00000000` and `-0` as `11111111`.
Here for N bit 1's complement representation we can represent numbers from `- (2^(N-1) - 1) to (2^(N-1) - 1)`.

### 2's complement

Given an 8 bit binary number `00010110`, its 2's complement is given by flipping its bits and adding 1 to it i.e. `11101010`.
The sign of the number can be identified using the MSB and here `0` has only one representation `00000000`.
Here for N bit 1's complement representation we can represent numbers from `-2^(N-1) to (2^(N-1) - 1)`.

## Sets using bits

A set over N elements can be represented by a N bit number, with 1 bit representing that the element is present and 0 representing it is absent.

`ALL_BITS` denotes the first N bits set to 1 (for N elements).

| Operation | Method |
|---|---|
| Set Union | A or B |
| Set Intersection | A & B |
| Set Subtraction | A & ~B |
| Set Negation | A ^ ALL_BITS |
| Set Bit | A or (1 << bit) |
| Clear Bit | A & (1 << bit) |
| Test Bit | A & (1 << bit) != 0 |

### Check if only one bit is set

Given x, we can check if only 1 bit set by checking if `x & (x - 1) == 0`.
`x = 0001000` then, `0001000 & 0000111 = 0`.
`x = 1101000` then, `1101000 & 1100111 = 1100000`.

### Find the maximum/minimum 1 bit in a set

We can just iterate from the highest/lowest bit and keep checking if a bit is set to 1. This seems to be a slow approach, but on average it is very fast. If each of the `2^N` sets are equally likely then the loop will only take 2 iterations on average.

We can also use the builtin CPU instructions to do this `__builtin_ctz()` and `__builtin_clz()` to do this. **Just make sure for argument 0 the function output is undefined.**

### Counting the number of 1 bits in a set

GCC has a builtin function `__builtin_popcount()` that does just this. This does not however translate to a hardware instruction like in the previous case but, even though it is still pretty efficient.

### Traversing all subsets

Traversing through all possible sets of N elements is trivial. One only needs to increment from `0 -> 2^N - 1` to generate all possible subsets. Also all sets come before their super sets making this extremely good for some DP solutions.

Traversing through all possible subsets of a given superset can be done in the reverse order. The iteration step would be `i -> (i - 1) & superset`, where `i = superset` in the beginning.

`11010 -> 11000 -> 10010 -> 10000 -> 01010 -> 01000 -> 00010 -> 00000`.

### Some common pitfalls

- When evaluating `a << b` or `b >> a` only the last 5 (6 in case of 64 bit integer) are used. Shifting right or left by 32 will do nothing.
- The operators `&` and `|` have lower precedence than boolean comparison operators hence `2 & 3 == 1` actually evaluates as `2 & (3 == 1)`,
- Also make sure to use **unsigned** integers if you want to utilise the topmost bit. On some compilers shift operations on negative values is not defined. Also in C++ the right shift operator inserts the MSB, for negative values it might have unforeseen consequences.

### Reversing the bits in an integer

```
1 x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);
2 x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);
```

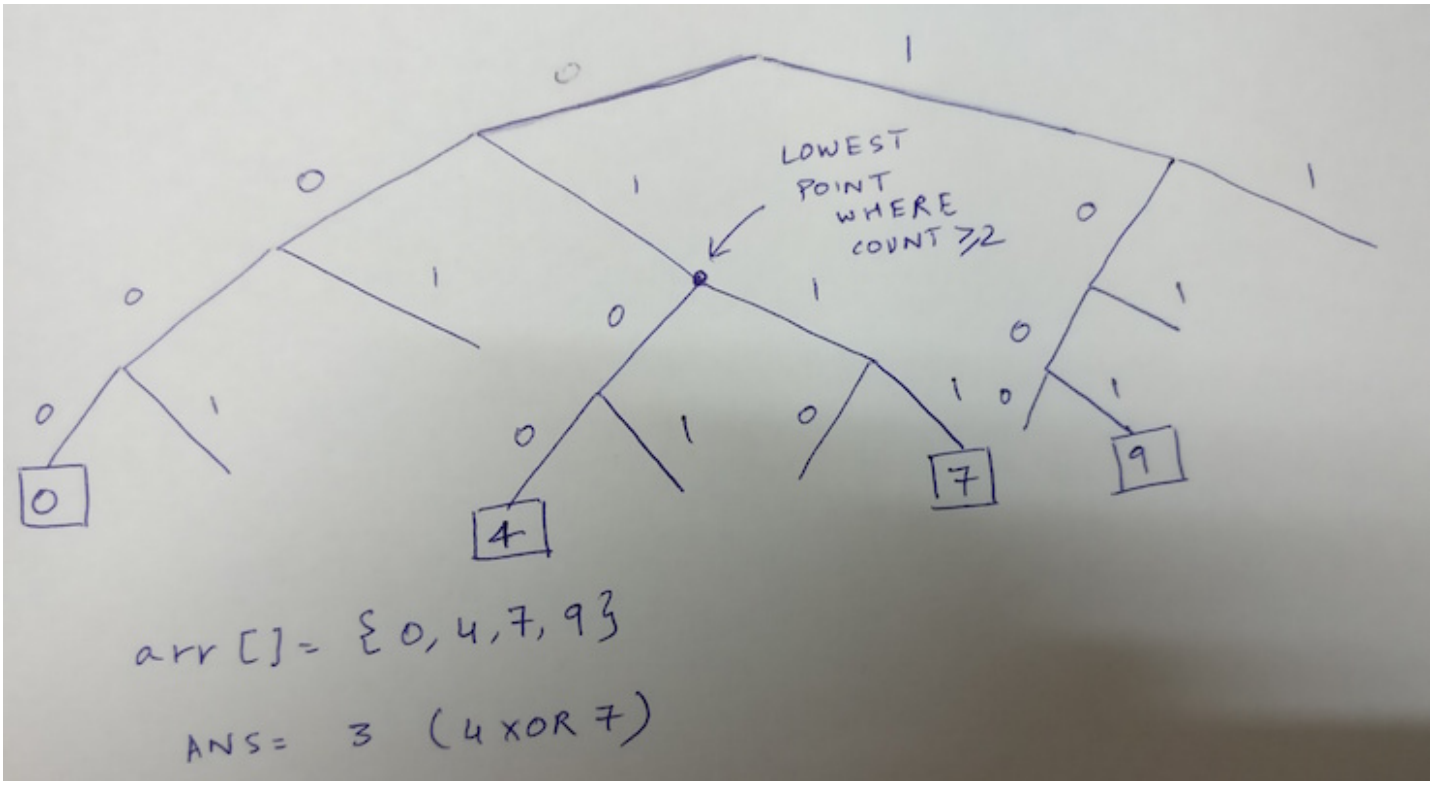
```
x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);
x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);
x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);
```

**Min XOR between a pair of numbers in an array**

Find a pair of numbers in a given array such that the XOR between them is minimum.

**Key Idea:** We need to prioritise eliminating the higher bits first because the sum of all lower bits is not equal to the higher bits.

Let's imagine a binary tree such where each N bit number is a unique leaf in the binary tree. We start at the root and for each significant bit we got the left (if it is 0) or right (if it is 1). Also at each node in the tree we maintain the count of all numbers below it. The minimum XOR value will be between a pair of numbers at the deepest level (because then the maximum higher bits would be same and hence, would cancel each other).




A quick observation shows that the leafs of this tree are in a sorted order. And the minimum value of XOR will always be between two successive indices in a sorted array.

**Solution:** Sort the array and just take XOR of consecutive element and return the minimum value.