filename: c-bit-manipulation-5pp-20250812.txt
https://cp-algorithms.com/algebra/bit-manipulation.html

Bit manipulation
December 20, 2024

Binary number
A binary number is a number expressed in the base-2 numeral system or binary numeral system, it is a method of mathematical expression which uses only two symbols: typically "0" (zero) and "1" (one).

We say that a certain bit is set, if it is one, and cleared if it is zero.

The binary number (a_k a_{k-1} ... a_1 a_0)_2 represents the number:
(a_k a_{k-1} ... a_1 a_0)_2 = a_k . 2^k + a_{k-1} . 2^{k-1} + ... + a_1 . 2^1 + a_0 . 2^0.

For instance the binary number 1101_2 represents the number 13:
1101_2 = 1 . 2^3 + 1 . 2^2 + 0 . 2^1 + 1 . 2^0
= 1 . 8 + 1 . 4 + 0 . 2 + 1 . 1 = 13

Computers represent integers as binary numbers. Positive integers (both signed and unsigned) are just represented with their binary digits, and negative signed numbers (which can be positive and negative) are usually represented with the Two's complement.

```
unsigned int unsigned_number = 13;
assert(unsigned_number == 0b1101);

int positive_signed_number = 13;
assert(positive_signed_number == 0b1101);

int negative_signed_number = -13;
assert(negative_signed_number == 0b1111'1111'1111'1111'1111'1111'1111'0011);
```

CPUs are very fast manipulating those bits with specific operations. For some problems we can take these binary number representations to our advantage, and speed up the execution time. And for some problems (typically in combinatorics or dynamic programming) where we want to track which objects we already picked from a given set of objects, we can just use an large enough integer where each digit represents an object and depending on if we pick or drop the object we set or clear the digit.

Bit operators
All those introduced operators are instant (same speed as an addition) on a CPU for fixed-length integers.

Bitwise operators
* & : The bitwise AND operator compares each bit of its first operand with the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
* | : The bitwise inclusive OR operator compares each bit of its first operand with the corresponding bit of its second operand. If one of the two bits is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
* ^ : The bitwise exclusive OR (XOR) operator compares each bit of its first operand with the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
* ~ : The bitwise complement (NOT) operator flips each bit of a number, if a bit is set the operator will clear it, if it is cleared the operator sets it.

Examples:

```
n         = 01011000
n-1       = 01010111
--------------------
n & (n-1) = 01010000

n         = 01011000
n-1       = 01010111
--------------------
n | (n-1) = 01011111

n         = 01011000
n-1       = 01010111
--------------------
n ^ (n-1) = 00001111

n         = 01011000
--------------------
~n        = 10100111
```

Shift operators
There are two operators for shifting bits.
* '>>' Shifts a number to the right by removing the last few binary digits of the number. Each shift by one represents an integer division by 2, so a right shift by k represents an integer division by 2^k.
E.g. 5 >> 2 = 101_2 >> 2 = 1_2 = 1 which is the same as 5/2^2 = 5/4 = 1.
For a computer though shifting some bits is a lot faster than doing divisions.
* '<<' Shifts a number to left by appending zero digits. In similar fashion to a right shift by k, a left shift by k represents a multiplication by 2^k.
E.g. 5 << 3 = 101_2 << 3 = 101000_2 = 40 which is the same as 5 . 2^3 = 5 . 8 = 40.
Notice however that for a fixed-length integer that means dropping the most left digits, and if

```
       you shift too much you end up with the number 0.

*** USEFUL TRICKS ***

Set/flip/clear a bit
   Using bitwise shifts and some basic bitwise operations we can easily set, flip or clear a bit. 1 << x is a
   number with only the x-th bit set, while ~(1 << x) is a number with all bits set except the x-th bit.
     * n | (1 << x)   sets the x-th bit in the number n
     * n ^ (1 << x)   flips the x-th bit in the number n
     * n & ~(1 << x)  clears the x-th bit in the number n

Check if a bit is set
   The value of the x-th bit can be checked by shifting the number x positions to the right, so that
   the x-th bit is at the unit place, after which we can extract it by performing a bitwise & with 1.

bool is_set(unsigned int number, int x) {
    return (number >> x) & 1;
}

Check if the number is divisible by a power of 2
   Using the 'and' operation, we can check if a number n is even because n & 1 = 0 if n is even,
   and n & 1 = 1 if n is odd. More generally, n is divisible by 2^k exactly when n & (2^k - 1) = 0.

bool isDivisibleByPowerOf2(int n, int k) {
    int powerOf2 = 1 << k;
    return (n & (powerOf2 - 1)) == 0;
}

   We can calculate 2^k by left shifting 1 by k positions. The trick works, because 2^k - 1 is a
   number that consists of exactly k ones. And a number that is divisible by 2^k must have zero
   digits in those places.

Check if an integer is a power of 2
   A power of two is a number that has only a single bit in it (e.g. 32 = 0010 0000_2)[test], while the predece
ssor
   of that number has that digit not set and all the digits after it set (31 = 0001 1111_2). So the bitwise
   AND of a number with it's predecessor will always be 0, as they don't have any common digits set. You can
   easily check that this only happens for the the power of twos and for the number $0$ which already has no
   digit set.

bool isPowerOfTwo(unsigned int n) {
    return n && !(n & (n - 1));
}

=== [test]===
$> p3
>>> for x in range(0,9):
...     print(f"2^{x} {2**x:<5}: {2**x:b}")
...
2^0 1    : 1
2^1 2    : 10
2^2 4    : 100
2^3 8    : 1000
2^4 16   : 10000
2^5 32   : 100000
2^6 64   : 1000000
2^7 128  : 10000000
2^8 256  : 100000000
>>>
=== [test]===

Clear the right-most set bit
   The expression n & (n-1) can be used to turn off the rightmost set bit of a number n. This
   works because the expression n-1 flips all bits after the rightmost set bit of n, including the
   rightmost set bit. So all those digits are different from the original number, and by doing a bitwise
   AND they are all set to 0, giving you the original number n with the rightmost set bit flipped.

   For example, consider the number 52 = 0011 0100_2:
n         = 00110100
n-1       = 00110011
--------------------
n & (n-1) = 00110000

Brian Kernighan's algorithm
   We can count the number of bits set with the above expression.

   The idea is to consider only the set bits of an integer by turning off its rightmost set bit (after
   counting it), so the next iteration of the loop considers the Next Rightmost bit.

int countSetBits(int n) {
    int count = 0;
    while (n) {
        n = n & (n - 1);
        count++;
    }
    return count;
```

```
}
```

Count set bits up to n

To count the number of set bits of all numbers upto the number n (inclusive), we can run the Brian Kernighan's algorithm on all numbers upto n. But this will result in a "Time Limit Exceeded" in contest submissions.

We can use the fact that for numbers upto 2^x (i.e. from 1 to 2^x - 1) there are x . 2^{x-1} set bits. This can be visualised as follows.

```
0 ->   0 0 0 0
1 ->   0 0 0 1
2 ->   0 0 1 0
3 ->   0 0 1 1
4 ->   0 1 0 0
5 ->   0 1 0 1
6 ->   0 1 1 0
7 ->   0 1 1 1
8 ->   1 0 0 0
```

We can see that the all the columns except the leftmost have 4 (i.e. 2^2) set bits each, i.e. upto the number 2^3 - 1, the number of set bits is 3 . 2^{3-1}.

With the new knowledge in hand we can come up with the following algorithm:

- Find the highest power of 2 that is lesser than or equal to the given number. Let this number be x.
- Calculate the number of set bits from 1 to 2^x - 1 by using the formula x . 2^{x-1}.
- Count the num of set bits in the most significant bit from 2^x to n and add it.
- Subtract 2^x from n and repeat the above steps using the new n.

```
int countSetBits(int n) {
    int count = 0;
    while (n > 0) {
        int x = std::bit_width(n) - 1;
        count += x << (x - 1);
        n -= 1 << x;
        count += n + 1;
    }
    return count;
}
```

Additional tricks

- n & (n + 1) clears all trailing ones: 0011 0111_2 --> 0011 0000_2.
- n | (n + 1) sets the last cleared bit: 0011 0101_2 --> 0011 0111_2.
- n & -n   extracts the last set bit: 0011 0100_2 --> 0000 0100_2.

Many more can be found in the book [https://en.wikipedia.org/wiki/Hacker's_Delight]Hacker's Delight.

Language and compiler support

C++ supports some of those operations since C++20 via the bit standard library:

- has_single_bit: checks if the number is a power of two
- bit_ceil / bit_floor: round up/down to the next power of two
- rotl / rotr: rotate the bits in the number
- countl_zero / countr_zero / countl_one / countr_one: count the leading/trailing zeros/ones
- popcount: count the number of set bits

Additionally, there are also predefined functions in some compilers that help working with bits. E.g. GCC defines a list at Built-in Functions Provided by GCC that also work in older versions of C++:

- __builtin_popcount(unsigned int) returns the number of set bits (__builtin_popcount(0b0001'0010'1100) == 4)
- __builtin_ffs(int) finds the index of the first (most right) set bit (__builtin_ffs(0b0001'0010'1100) == 3)
- __builtin_clz(unsigned int) the count of leading zeros (__builtin_clz(0b0001'0010'1100) == 23)
- __builtin_ctz(unsigned int) the count of trailing zeros (__builtin_ctz(0b0001'0010'1100) == 2)
- __builtin_parity(x) the parity (even or odd) of the number of ones in the bit representation

Note that some of the operations (both the C++20 functions and the Compiler Built-in ones) might be quite slow in GCC if you don't enable a specific compiler target with #pragma GCC target("popcnt").
