Bitwise tips:

Left shift (<<):

- shifting left is equivalent to multiplication by power of 2, for example:
  - $6 << 1 = 6 \times 2$
  - $6 << 3 = 6 \times (2^3) = 6 \times 8 = 48$

Right shift (>>):

- shifting right is equivalent to division by power of 2, for example:
  - $12 >> 1 = 12 : (2^1) = 6$
  - $12 >> 2 = 12 : (2^2) = 3$

$$x = \sum_{k=0}^{w-1} x_k 2^k \quad \text{(unsigned integer)}$$

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1} \quad \text{(sign integer)}$$

sign bit

- Unsigned integer, example:

8 bit word 0b10010110 represents the unsigned value, $150 = 2+4+16+128$

- Signed integer (two's complement), example:

8 bit word 0b10010110 represents the signed value $-106 = 2+4+16-128$

- We have $0b00\ldots0 = 0$
- what is the value of $x = 0b11\ldots1$?

$$x = \left(\sum_{k=0}^{w-2} x_k 2^k \right) - x_{w-1} 2^{w-1}$$

$$= \left(\sum_{k=0}^{w-2} 2^k \right) - 2^{w-1}$$

$$= (2^{w-1} - 1) - 2^{w-1}$$

$$= -1$$

Complementary Relationship

$$x + \sim x = -1 \Rightarrow -x = \sim x + 1$$

Exp:

$x = 0b0110\,1100\,0$

$\sim x = 0b1001\,0011\,1$

$-x = 0b1001\,0100\,0$

* The prefix "0b" designates a Boolean constant.

To translate from hex to binary, translate each hex digit to its binary equivalent, and concatenate the bits.

Example: 0x DEC1DE2C0DE4F00D is
-> the prefix 0x designates a hex constant

1101 1110 1100 0001 1101 1110 0010 1100 0000 1101 1110 0100

D E C 1 D E 2 C 0 D E 4

1111 0000 0000 1101

F 0 0 D

Bitwise operators:

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	Shift left
>>	Shift right

- Examples (8-bit word)

A= 0b10110011
B= 0b01101001
A&B= 0b00100001
A|B= 0b11111011
A^B= 0b11011010
~A= 0b01001100
A>>3= 0b00010110
A<<2= 0b11001100

PROBLEMS:

1. Set kth bit in a word x to 1.

- IDEA:
  * SHIFT and OR.

$$y = x \mid (1 << k)$$

- EXAMPLE:

$k = 7$

7th

x	10111101**0**11011
$1 << k$	00000000**1**00000
$x \mid (1 << k)$	10111101**1**11011

2. Clear the kth bit in a word x.

- IDEA:
  * SHIFT, complement, and AND

$$y = x \,\&\, \sim(1 << k)$$

- EXAMPLE:

$k = 7$

-> 7th bit

x	10111101[1]10110101
1<<k	00000000100000000
~(1<<k)	11111111011111111
x & ~(1<<k)	10111101011010101

3. Toggle / Flip the kth bit in a word x

- IDEA:
  * SHIFT and XOR

$$y = x \wedge (1 << k)$$

- EXAMPLE (0 -> 1):

$k = 7$

x	10111101[0]11010101 ← kth bit
1<<k	00000000100000000
x ^ (1<<k)	10111101111010101

4. Extract a bit field from a word x

- IDEA:
  * mask and shift

(x & mask) >> shift

- EXAMPLE:

shift = 7

x	10111101011011 01
mask	00000111 1100 0000
x & mask	00000101 0000 0000
x & mask >> shift	0000000000000101 0

- this is a good trick to know if you're working with compressed or encoded data.

5. Set a bit field in a word x to a value y.

- IDEA:

* invert mask to clear, and OR the shifted value

$$x = (x \,\&\, \sim mask) \mid (y << shift)$$

- EXAMPLE:

shift = 7 For safety's sake, ((y<<shift)&mask)

x	10111101011011 01
y	0000000000000011
mask	00000111 1100 0000
x & ~mask	10111000 1101 1101
(x & ~mask) | (y << shift)	10111001 1111 0111

6. Ordinary swap: swap two integers x and y.

- the standard way of doing this is to use a temporary variable:

```
t = x
x = y
y = t
```

* we can do this by using bit tricks, swapping x and y without using a temporary.

```
x = x ^ y   -> Mask with 1's where bits differ.
y = x ^ y   -> Flip the bits in y that differ from x.
x = x ^ y   -> Flip the bits in x that differ from y.
```

For example:

x	10111101	10010011	10010011	00101110
y	00101110	00101110	10111101	10111101

- why it works:

XOR is its own inverse:

$(x \wedge y) \wedge y \Rightarrow x$

x	y	x ^ y	(x ^ y) ^ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

* when we XOR something twice, it just cancel out and we get back the original thing.

- Performance:
  Poor at exploiting „instruction-level parallelism (ILP)"! ("the naive approach is faster).

7. Find the minimum 'z' of two integers x and y.
   The standard approach will be:

```
if (x < y):
    z = x
else:
    z = y
```

- Performance:

• one performance problem with this code is that there is a branch in this code (modern machines will do branch prediction, and for whatever branch predicts the code to take it's going to do prefetching and execute some of the instructions in advance). But the problem is, if it mispredicts the branch, it does a lot of wasted work, and the processor has to empty the pipeline and undo all of the work that it did.

- Caveat:

• the compiler is usually smart enough to optimize away the unpredictable branch, but maybe not.

? Is there a way to do a minimum without using a branch

$$r = y \wedge ((x \wedge y) \,\&\, -(x < y))$$

Why it works:

- The C language represents the Booleans TRUE and FALSE with the integers 1 and 0.

- If $x < y$, then $-(x < y) \Rightarrow -1$, which is all 1's in two's complement representation. Therefore, we have $y \wedge (x \wedge y) \Rightarrow x$

- If $x \geq y$, then $-(x < y) \Rightarrow 0$. Therefore, we have $y \wedge 0 \Rightarrow y$.

Info! Modern compilers can perform this optimization better than you can. (the branchless version is usually slower than the branching).

* Why learn bit hacks if they don't even work?

- Because the compiler does them, and it will help to understand what the compiler is doing when you look at the assembly code.
- Because sometimes the compiler doesn't optimize, and you have to do it yourself by hand.
- Because many bit hacks for words extend naturally to bit and word hacks for vectors.
- Because these tricks arise in other domains.
- Because they're fun! YESSS :)

8. Modular Addition: compute $(x+y) \bmod n$, assuming that $0 \le x < n$ and $0 \le y < n$.

$r = (x+y) \% n$ (division is expensive, unless by a power of 2)

$z = x + y$
$r = z$ if $z < n$ else $z - n$ (Unpredictable branch is expensive)

$z = x + y$
$r = z - (n \& -(z >= n))$ → same trick as minimum

9. Round up to a Power of 2

Compute $2^{\lg n}$ → Notation $\lg n = \log_2 n$

uint64_t n; Bit $\lceil \lg n \rceil - 1$

: (we flip the rightmost 1 with 0, and rest with 1)

--n; (decrement n)

n |= n >> 1 ∧

Example:

0010000000100000

0010000000100111

0011000000110111

```
n |= n >> 2
n |= n >> 4
n |= n >> 8
n |= n >> 16
n |= n >> 32
```

Populate all bits to the right with 1

0011100 0111111
0111111 1111111
⋮
01000000000000000

++n → (we add one to that) → set bit $\lceil \lg n \rceil$

Why decrement?

- if n is already a power of 2 and if we don't decrement n, this isn't going to work because the $\log n - 1$ bit isn't set. But if we decrement n, then it's going to guarantee us that the $\log n - 1$ bit is set so that we can propagate that to the right.

10. Compute the mask of the least-significant 1 in word x.

- EXAMPLE: we want a mask that has a 1 in only the position of the least significant 1 in x, and 0's everywhere else.

$r = x \,\&\, (-x)$

- Exp:

x	00100000010010000
$-x$	11011111101110000
$x \,\&\, (-x)$	00000000000010000

- Why it works
  - the binary representation of $-x$ is $(\sim x)+1$

- Question
  How do you find the index of the bit, $\lg x$?

11. Log Base 2 of a Power of 2:

Compute $\lg x$, where x is a power of 2

```
const uint64_t deBruijn = 0x022fdd63cc95386d
const int convert[64] = {
```

K=8

```
 0,  1,  2, 53,  3,  7, 54, 27,
 4, 38, 41,  8, 34, 55, 48, 28,
62,  5, 39, 46, 44, 42, 22,  9,
24, 35, 59, 56, 49, 18, 29, 11,
```

```
    63, 52,  6, 26, 37, 40, 33, 47,
    61, 45, 43, 21, 23, 58, 17, 10,
    51, 25, 36, 32, 60, 20, 57, 16,
    50, 31, 19, 15, 30, 14, 13, 12
};
r = convert[(x * deBruijn) >> 58]
```

- Why it works

A <u>de Bruijn sequence</u> s of length 2^k is a cyclic 0-1 sequence such that each of the 2^k 0-1 strings of length k occurs exactly once as a substring of s.

- Example: $k = 3$

All of the 8 possible substrings of length 3

0 0 0 1 1 1 0 1

```
0   0 0 0
1     0 0 1
2       0 1 1
3         1 1 1
4           1 1 0
5             1 0 1
6               0 1 0
```

← start with all 0's

```
const int convert[8] = {0,1,6,2,7,5,4,3};
```

For example: (this is the same as left shifting)

0b00011101 * 2^4 => 0b11010000

0b11010000 >>5 => 6

convert[6] => 4

110 appears at the beginning of the sequence

we want to extract this out, and we right shifting five position => 110 = 6

Performance

Limited by multiply and table look up.

12. Population Count 1:

Count the number of 1 bits in a word x.

```
for (r=0; x!=0; ++r)
    x &= x - 1
```

Repeatedly eliminate the least-significant 1.

Example:

x	00101101110010000
$x-1$	00101101110011111
$x \& (x-1)$	00101101110000000

Issue:

- fast if the population count is small, running time is proportional to the number of bits in the word.

13. Population Count II:

```
static const int count[256] =
{0, 1, 1, 2, 1, 2, 2, 3, 1, ..., 8};
for(int k=0; x!=0; x>>=8)
    k += count[x & 0xFF];
```

$x^k = \underbrace{x x \cdots x}_{k \text{ times}}$

14. Population Count III:

```
// Create masks
M5 = ~((-1)<<32);   // 0^32,1^32
M4 = M5 ^ (M5<<16); // (0^16,1^16)^2
M3 = M4 ^ (M4<<8);  // (0^8,1^8)^4
M2 = M3 ^ (M3<<4);  // (0^4,1^4)^8
M1 = M2 ^ (M2<<2);  // (0^2,1^2)^16
```

```
M0 = M1 ^ (M1 << 1); // (01)^32

// Compute pop count                    Avoid overflow
x = ((x >> 1) & M0) + (x & M0)
x = ((x >> 2) & M1) + (x & M1)
x = ((x >> 4) + x) & M2
x = ((x >> 8) + x) & M3                 No worry about overflow
x = ((x >> 16) + x) & M4
x = ((x >> 32) + x) & M5
```

Performance: $O(\lg w)$ time, where w = word length