Given a positive integer, output its complement number. The complement strategy is to flip the bits of its binary representation.

**Note:**

1. The given integer is guaranteed to fit within the range of a 32-bit signed integer.
2. You could assume no leading zero bit in the integer’s binary representation.

**Example 1:**

**Input:** 5

**Output:** 2

**Explanation:** The binary representation of 5 is 101 (no leading zero bits), and its complement is 010. So you need to output 2.

**Example 2:**

**Input:** 1

**Output:** 0

**Explanation:** The binary representation of 1 is 1 (no leading zero bits), and its complement is 0. So you need to output 0.

**Solution:** Find the highest bit set. Then start flipping all the bits. Flipping all the bits can be done in O(1) time using masks. Let’s say highest bit set was on *nth* bit (bit numbering starting from 1), then (1 ≪ (n)) – 1) will give us the mask. Now, use the mask and xor it with the original number.

To find the highest bit set, a detailed algorithm has been discussed in the Trick document. A copy is also pasted below. The below algorithm find the rounded 2 power value of the number given (which is what we want and we’ll subtract 1 to get the mask).

**Rounding up to next power of 2.**

Source. <http://graphics.stanford.edu/~seander/bithacks.html#RoundUpPowerOf2>

The naïve method, is the find the highest bit set, then the bit higher (only that) to that would be the rounded two powers. E.g., if number is 5, i.e., 101, highest bit set is 3rd bit. Now, if we set the fourth bit to 1 and no other bit, we get 1000, which is 8 (that is, rounded to next power of 2).

Smart solution given by the link above, is to set all the bits from the highest bit set till first bit to 1, then add 1 to them—which makes them rounded to next 2 power number. For example, binary 111 + 1 is 1000 which is 8. I.e., 7’s next 2 power number is 8. How to achieve this faster? By recursively setting first highest bit, then first two highest bits, then 4 highest bits etc. Example, let’s say we have a number with highest bit set like this: 0001XXXXXX. Here X means it can be zero or one (doesn’t matter). For peace sake, let’s consider that number as 0001010001.

Now, first iterations

0001XXXXXX  
OR 00001XXXXX (Right shift by 1)  
= 00011XXXXX

Second Iteration

00011XXXXX  
OR 0000011XXX (Right shift by 2)  
= 0001111XXX

Third Iteration

0001111XXX  
OR 0000000111 (Right shift by 4)  
= 0001111111

Note how 0001010001 is transformed to 0001111111. All we need to do now is to add 1 to it. This makes the number 0010000000 which is the next 2 powers of the given number. This algorithm works by using the property that any number, 1 less than two powers has all its bit set. This algorithm works in *log(bits)*. I.e., number of bits used to represent the number. Code which does this is below.

unsigned int v; // compute the next highest power of 2 of 32-bit v

v--;

v |= v >> 1;

v |= v >> 2;

v |= v >> 4;

v |= v >> 8;

v |= v >> 16;

v++;

**Second solution:** This also runs ins **log(bits)** time. Find the highest bit set using kind of binary search. Divide the number into two halvs, check if the first half is zero (meaning highest bit is the second half), if yes then goto second half, else recurse on the first half. When there are only two bits left, we’re at the base case.

int round\_two\_powers(int num) {

int divisor = 16;

int bit\_count = 0;

*/\* Doing binary search here \*/*

while (divisor >= 1) {

*/\* Check if the first part of number is greater than zero.*

*\* If yes, this means the highest set bit is in the first half \*/*

if (num >> divisor) {

*/\* as highest set bit is in the second half, we would need divisor number of*

*\* bit shifts to get to first half \*/*

bit\_count += divisor;

*/\* We need to recurse in the first half again, thus reduce the num to only*

*\* first half \*/*

num >>= divisor;

} else {

*/\* second half. As first half is zero, the highest bit should be in*

*\* the second half\*/*

}

*/\* We keep halving the division of bits. I.e., we divide the number into 16 bit two chunks.*

*\* Then we divide it into two 8-bit chunks the, two 4-bits chunks till two 1-bits cunks*

*\*/*

divisor /= 2;

}

*/\* bit\_count will tell the exact highest bit set. We need one bit more to that \*/*

return 1 << (bit\_count + 1);

}