231. Power of two

Aug. 17, 2019 | 20.4K views

Previous Next

Given an integer, write a function to determine if it is a power of two.

Example 1:

```
Input: 1
Output: true
Explanation: 2^0 = 1
```

Example 2:

```
Input: 16
Output: true
Explanation: 2^4 = 16
```

Example 3:

```
Input: 218
Output: false
```

Solution

Overview

We're not going to discuss here an obvious $\mathcal{O}(\log N)$ time solution

Java | Python

```
class Solution(object):
    def isPowerOfTwo(self, n):
        if n == 0:
            return False
        while n % 2 == 0:
            n /= 2
        return n == 1
```

Instead, the problem will be solved in $\mathcal{O}(1)$ time with the help of bitwise operators. The idea is to discuss such bitwise tricks as

- How to get / isolate the rightmost 1-bit : `x & (-x)`.
- How to turn off (= set to 0) the rightmost 1-bit : `x & (x - 1)`.

These tricks are often used as something obvious in more complex bit-manipulation solutions, like for N Queens problem, and it's important to recognize them to understand what is going on.

Intuition

The idea behind both solutions will be the same: a power of two in binary representation is one 1-bit, followed by some zeros:

$1 = (00000001)_2$

$2 = (00000010)_2$

$4 = (00000100)_2$

$8 = (00001000)_2$

A number which is not a power of two, has more than one 1-bit in its binary representation:

$3 = (00000011)_2$

$5 = (00000101)_2$

$6 = (00000110)_2$

$7 = (00000111)_2$

The only exception is 0, which should be treated separately.

Approach 1: Bitwise Operators : Get the Rightmost 1-bit

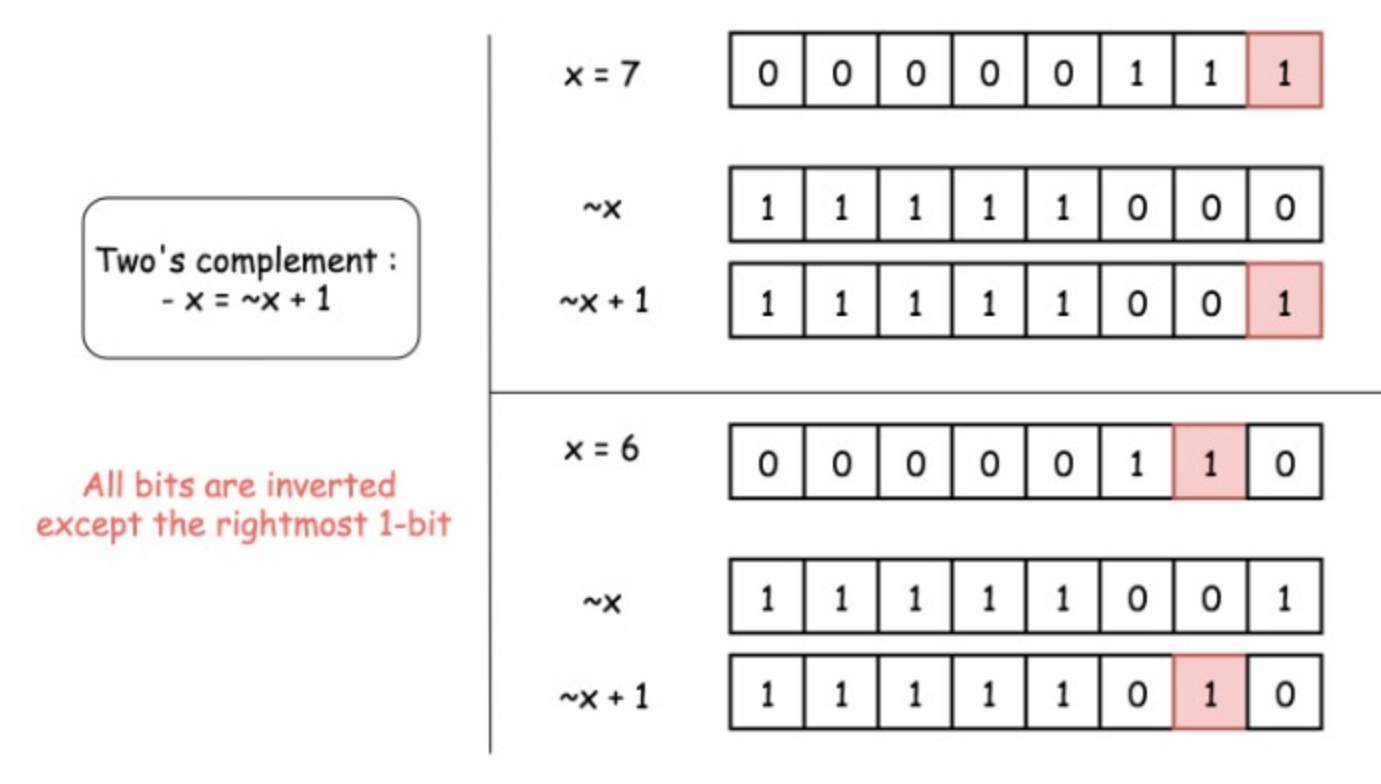
Get/Isolate the Rightmost 1-bit

Let's first discuss why `x & (-x)` is a way to keep the rightmost 1-bit and to set all the other bits to 0.

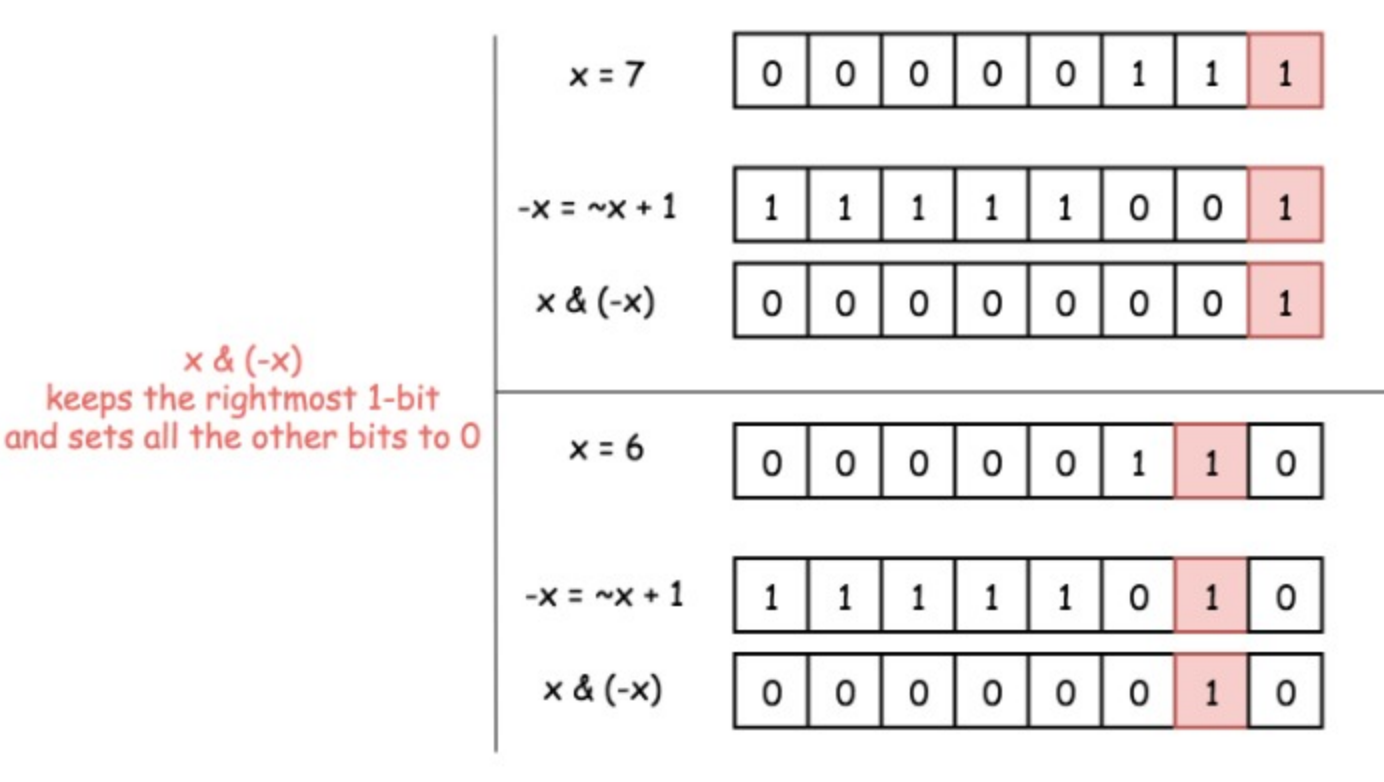
Basically, that works because of two's complement. In two's complement notation $-x$ is the same as $\neg x + 1$. In other words, to compute $-x$ one has to revert all bits in x and then to add 1 to the result.

Adding 1 to $\neg x$ in binary representation means to carry that 1-bit till the rightmost 0-bit in $\neg x$ and to set all the lower bits to zero. Note, that the rightmost 0-bit in $\neg x$ corresponds to the rightmost 1-bit in x.

> In summary, $-x$ is the same as $\neg x + 1$. This operation reverts all bits of x except the rightmost 1-bit.



Hence, x and -x have just one bit in common - the rightmost 1-bit. That means that `x & (-x)` would keep that rightmost 1-bit and set all the other bits to 0.

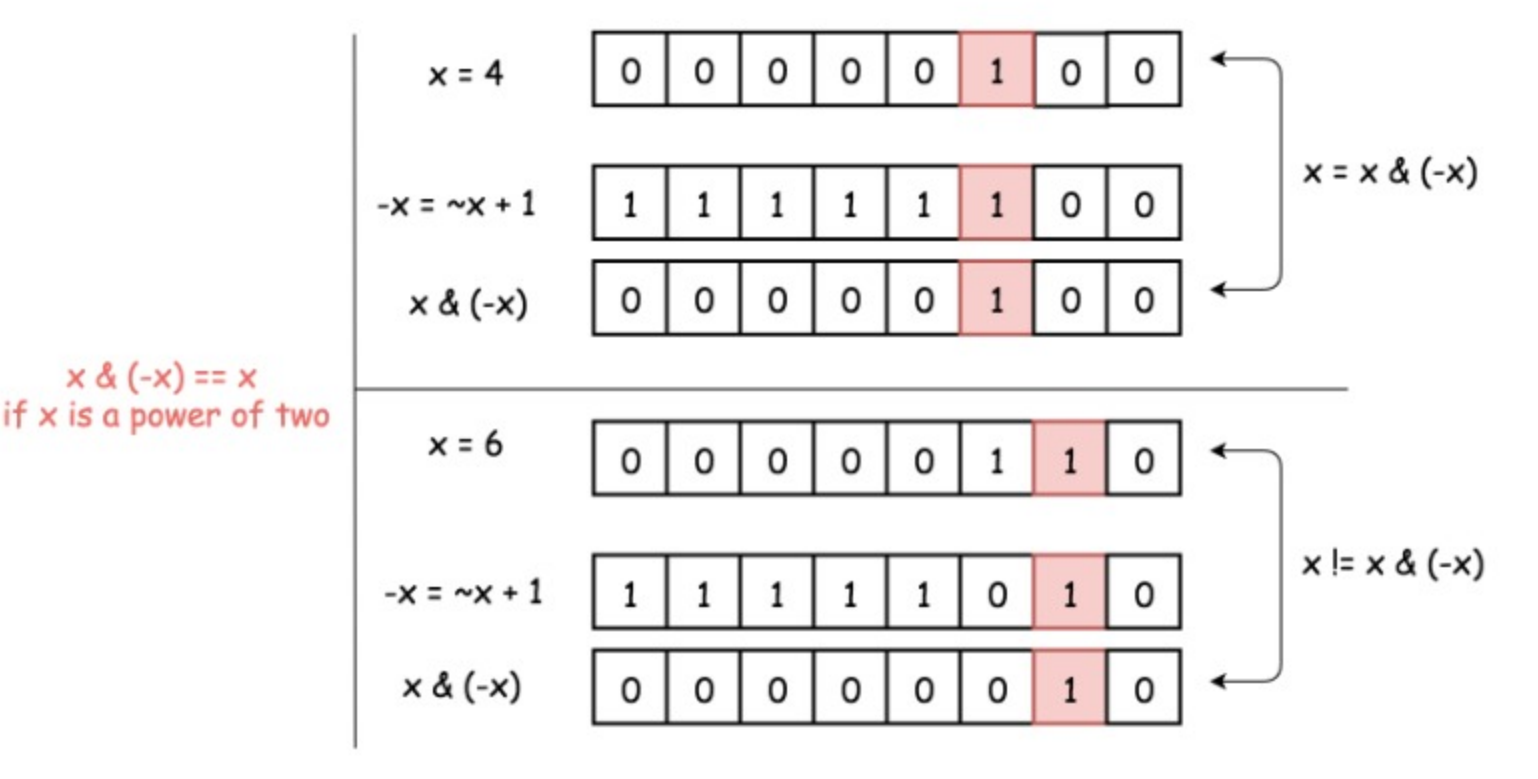


Detect Power of Two

So let's do `x & (-x)` to keep the rightmost 1-bit and to set all the others bits to zero. As discussed above, for the power of two it would result in `x` itself, since a power of two contains just one 1-bit.

Other numbers have more than 1-bit in their binary representation and hence for them `x & (-x)` would not be equal to `x` itself.

Hence a number is a power of two if `x & (-x) == x`.



Implementation

C++ | Java | Python | C

```
class Solution(object):
    def isPowerOfTwo(self, n):
        if n == 0:
            return False
        return n & (-n) == n
```

Complexity Analysis

- Time complexity : $\mathcal{O}(1)$.
- Space complexity : $\mathcal{O}(1)$.

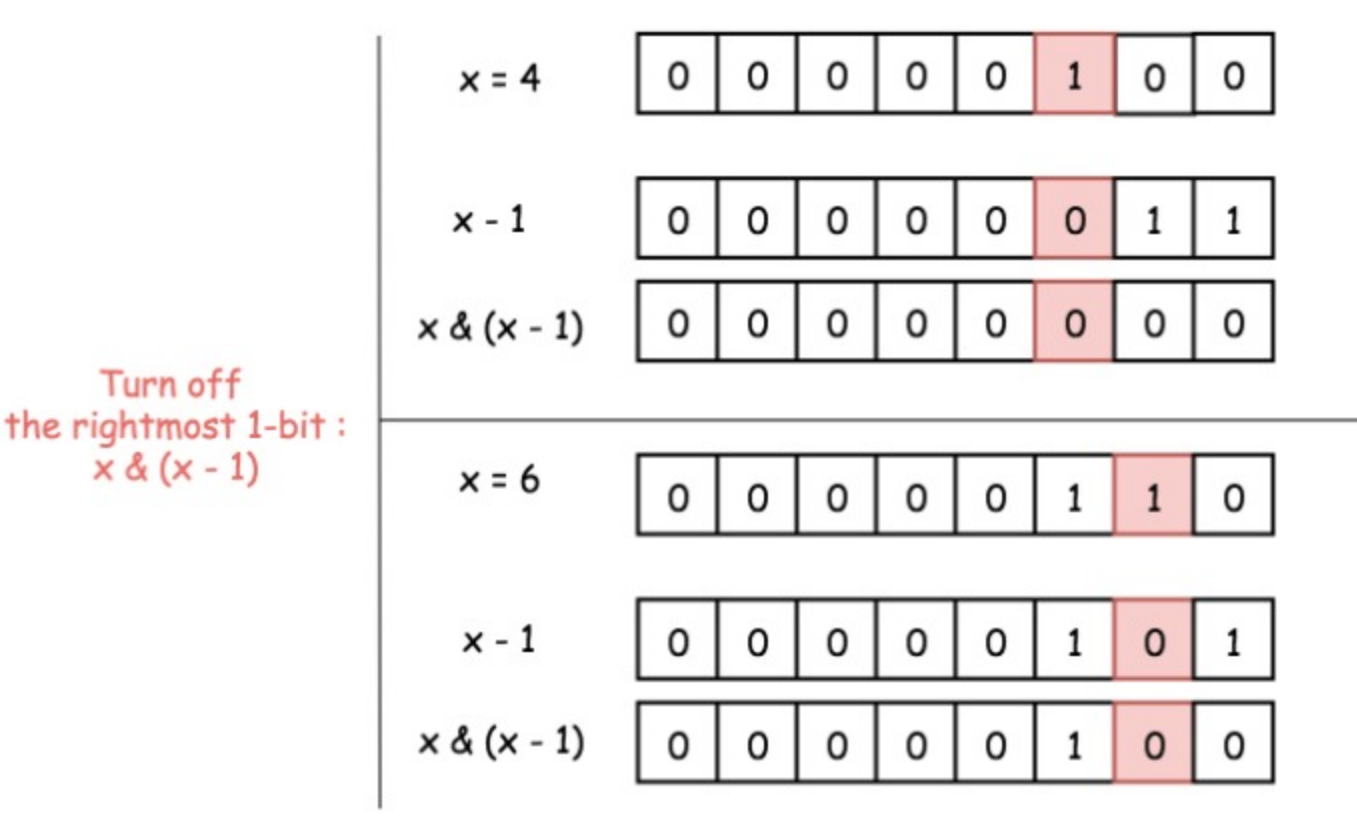
Approach 2: Bitwise operators : Turn off the Rightmost 1-bit

Turn off the Rightmost 1-bit

Let's first discuss why `x & (x - 1)` is a way to set the rightmost 1-bit to zero.

To subtract 1 means to change the rightmost 1-bit to 0 and to set all the lower bits to 1.

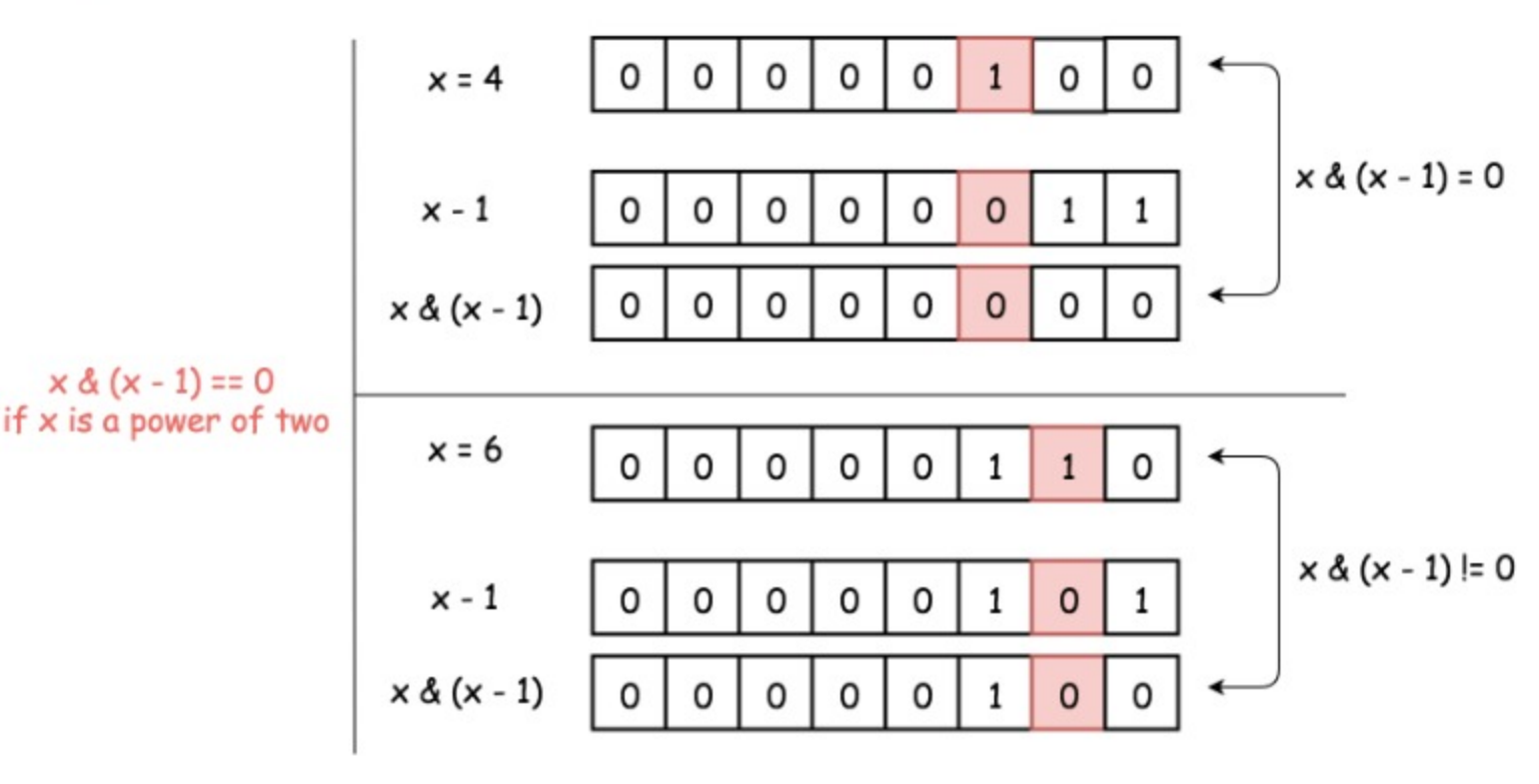
Now AND operator: the rightmost 1-bit will be turned off because `1 & 0 = 0`, and all the lower bits as well.



Detect Power of Two

The solution is straightforward:

1. Power of two has just one 1-bit.
2. `x & (x - 1)` sets this 1-bit to zero, and hence one has to verify if the result is zero `x & (x - 1) == 0`.



Implementation

C++ | Java | Python | C

```
class Solution(object):
    def isPowerOfTwo(self, n):
        if n == 0:
            return False
        return n & (n - 1) == 0
```

Complexity Analysis

- Time complexity : $\mathcal{O}(1)$.
- Space complexity : $\mathcal{O}(1)$.

Previous Next