Previous Next

190. Reverse Bits

March 23, 2020 | 14.9K views

Average Rating: 3.43 (21 votes)

Reverse bits of a given 32 bits unsigned integer.

Example 1:

```
Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string 00000010100101000001111010011100 represents the u
```

Example 2:

```
Input: 11111111111111111111111111111101
Output: 10111111111111111111111111111111
Explanation: The input binary string 11111111111111111111111111111101 represents the u
```

Note:

- Note that in some languages such as Java, there is no unsigned integer type. In this case, both input and output will be given as signed integer type and should not affect your implementation, as the internal binary representation of the integer is the same whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above the input represents the signed integer `-3` and the output represents the signed integer `-1073741825`.

Follow up:

If this function is called many times, how would you optimize it?

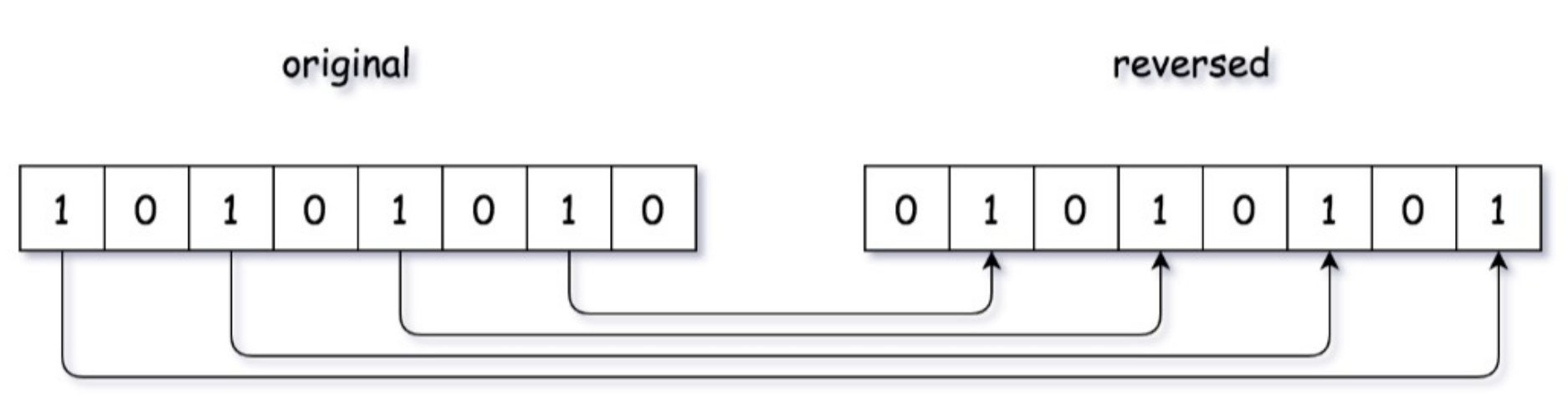
Solution

Approach 1: Bit by Bit

Intuition

Though the question is not difficult, it often serves as a warm-up question to kick off the interview. The point is to test one's basic knowledge on data type and bit operations.

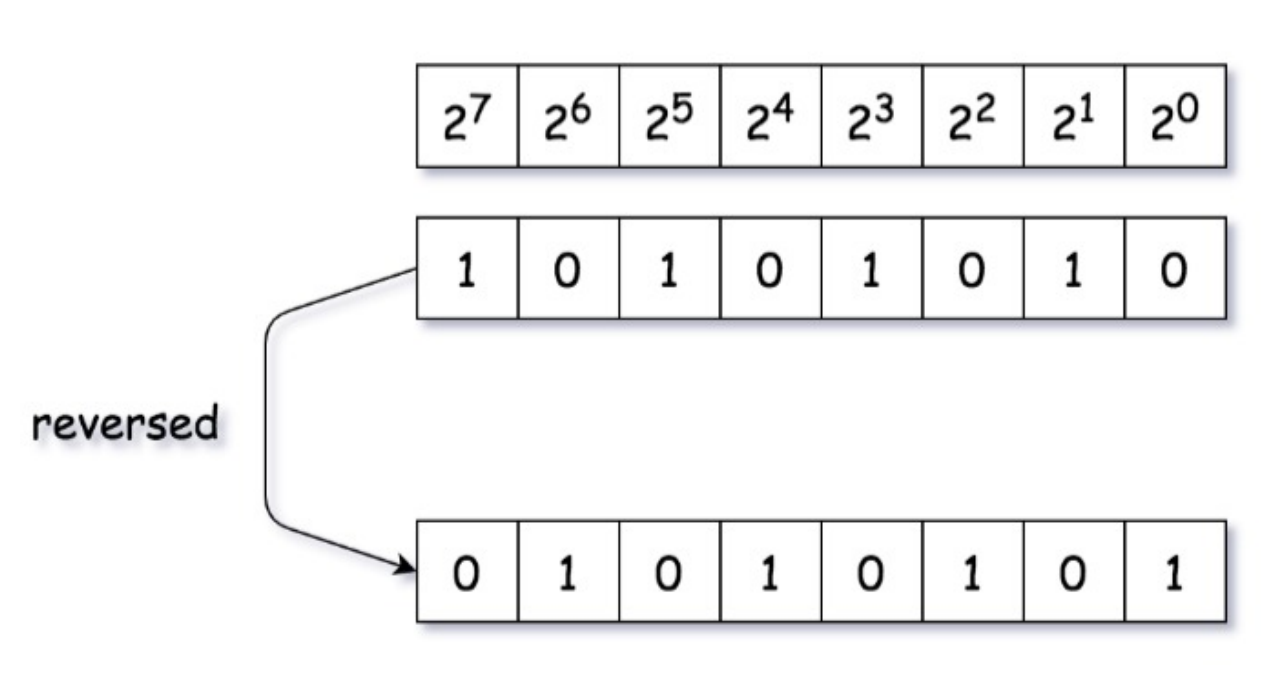
> As one of the most intuitive solutions that one could come up during an interview, one could reverse the bits **one by one**.



As easy as it sounds, the above intuition could lead to quite some variants of implementation. For instance, to retrieve the *right-most* bit in an integer `n`, one could either apply the modulo operation (*i.e.* `n % 2`) or the bit AND operation (*i.e.* `n & 1`). Another example would be that in order to combine the results of reversed bits (*e.g.* $2^a, 2^b$), one could either use the addition operation (*i.e.* $2^a + 2^b$) or again the bit OR operation (*i.e.* $2^a | 2^b$).

Algorithm

Here we show on example of implementation based on the above intuition.



> The key idea is that for a bit that is situated at the index `i`, after the reversion, its position should be `31-i` (note: the index starts from zero).

- We iterate through the bit string of the input integer, from right to left (*i.e.* `n = n >> 1`). To retrieve the right-most bit of an integer, we apply the bit AND operation (`n & 1`).
- For each bit, we reverse it to the correct position (*i.e.* `(n & 1) << power`). Then we accumulate this reversed bit to the final result.
- When there is no more bits of one left (*i.e.* `n == 0`), we terminate the iteration.

C++ | Python | Go — Copy

```cpp
class Solution {
  public:
  uint32_t reverseBits(uint32_t n) {
    uint32_t ret = 0, power = 31;
    while (n != 0) {
      ret += (n & 1) << power;
      n = n >> 1;
      power -= 1;
    }
    return ret;
  }
};
```

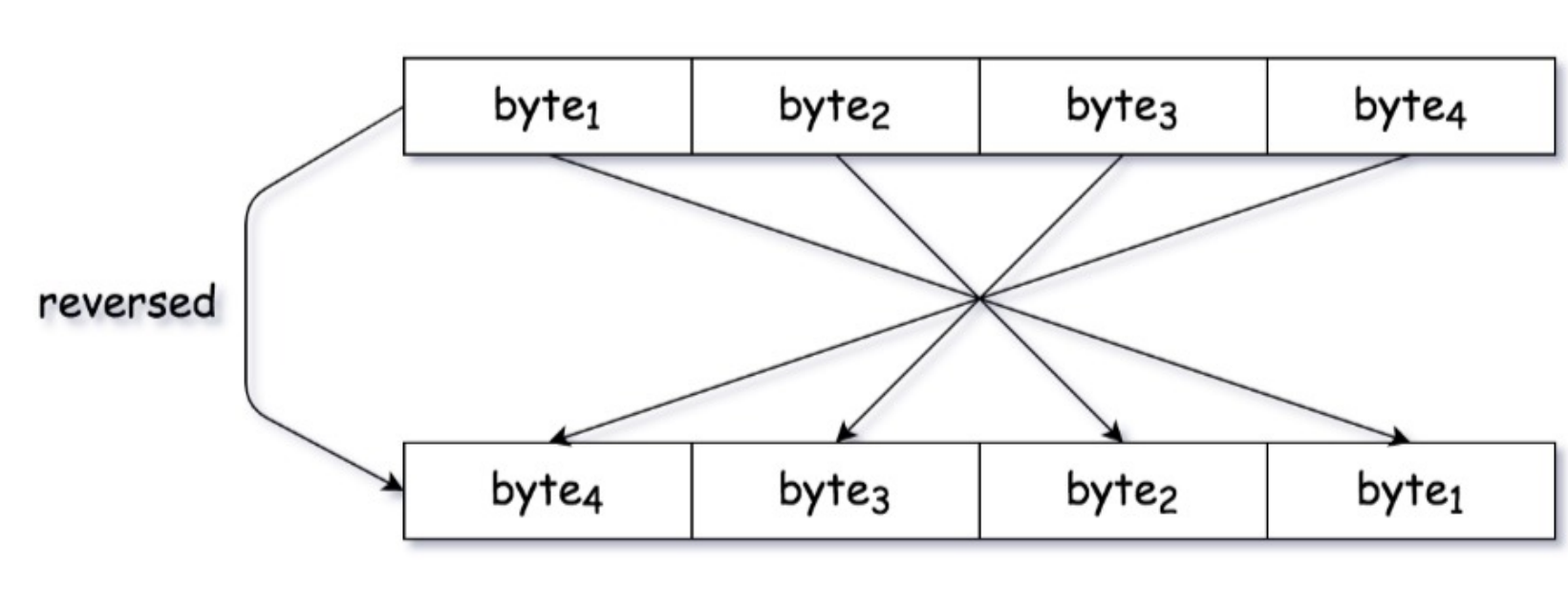
Complexity

- Time Complexity: $\mathcal{O}(1)$. Though we have a loop in the algorithm, the number of iteration is fixed regardless the input, since the integer is of fixed-size (32-bits) in our problem.
- Space Complexity: $\mathcal{O}(1)$, since the consumption of memory is constant regardless the input.

Approach 2: Byte by Byte with Memoization

Intuition

Someone might argument it might be more efficient to reverse the bits, **per byte**, which is an unit of 8 bits. Though it is not necessarily true in our case, since the input is of fixed-size 32-bit integer, it could become more advantageous when dealing with the input of long bit stream.



Another implicit advantage of using **byte** as the unit of iteration is that we could apply the technique of **memoization**, which caches the previously calculated values to avoid the re-calculation.

The application of memoization can be considered as a response to the **follow-up** question posed in the description of the problem, which is stated as following:

> *If this function is called many times, how would you optimize it?*

To reverse bits for a byte, one could apply the same algorithm as we show in the above approach. Here we would like to show a different algorithm which is solely based on the arithmetic and bit operations without resorting to any loop statement, as following:

```python
def reverseByte(byte):
    return (byte * 0x0202020202 & 0x010884422010) % 1023
```

The algorithm is documented as "reverse the bits in a byte with 3 operations" on the online book called **Bit Twiddling Hacks** by Sean Eron Anderson, where one can find more details.

Algorithm

- We iterate over the bytes of an integer. To retrieve the right-most byte in an integer, again we apply the bit AND operation (*i.e.* `n & 0xff`) with the bit mask of `11111111`.
- For each byte, first we reverse the bits within the byte, via a function called `reverseByte(byte)`. Then we shift the reversed bits to their final positions.
- With the function `reverseByte(byte)`, we apply the technique of memoization, which caches the result of the function and returns the result directly for the future invocations of the same input.

Note that, one could opt for a smaller unit rather than byte, *e.g.* a unit of 4 bits, which would require a bit more calculation in exchange of less space for cache. It goes without saying that, the technique of memoization is a trade-off between the space and the computation.

C++ | Python | Go | Python3 — Copy

```python
import functools

class Solution:
    # @param n, an integer
    # @return an integer
    def reverseBits(self, n):
        ret, power = 0, 24
        while n:
            ret += self.reverseByte(n & 0xff) << power
            n = n >> 8
            power -= 8
        return ret

    # memoization with decorator
    @functools.lru_cache(maxsize=256)
    def reverseByte(self, byte):
        return (byte * 0x0202020202 & 0x010884422010) % 1023
```

Complexity

- Time Complexity: $\mathcal{O}(1)$. Though we have a loop in the algorithm, the number of iteration is fixed regardless the input, since the integer is of fixed-size (32-bits) in our problem.
- Space Complexity: $\mathcal{O}(1)$. Again, though we used a cache keep the results of reversed bytes, the total number of items in the cache is bounded to $2^8 = 256$.

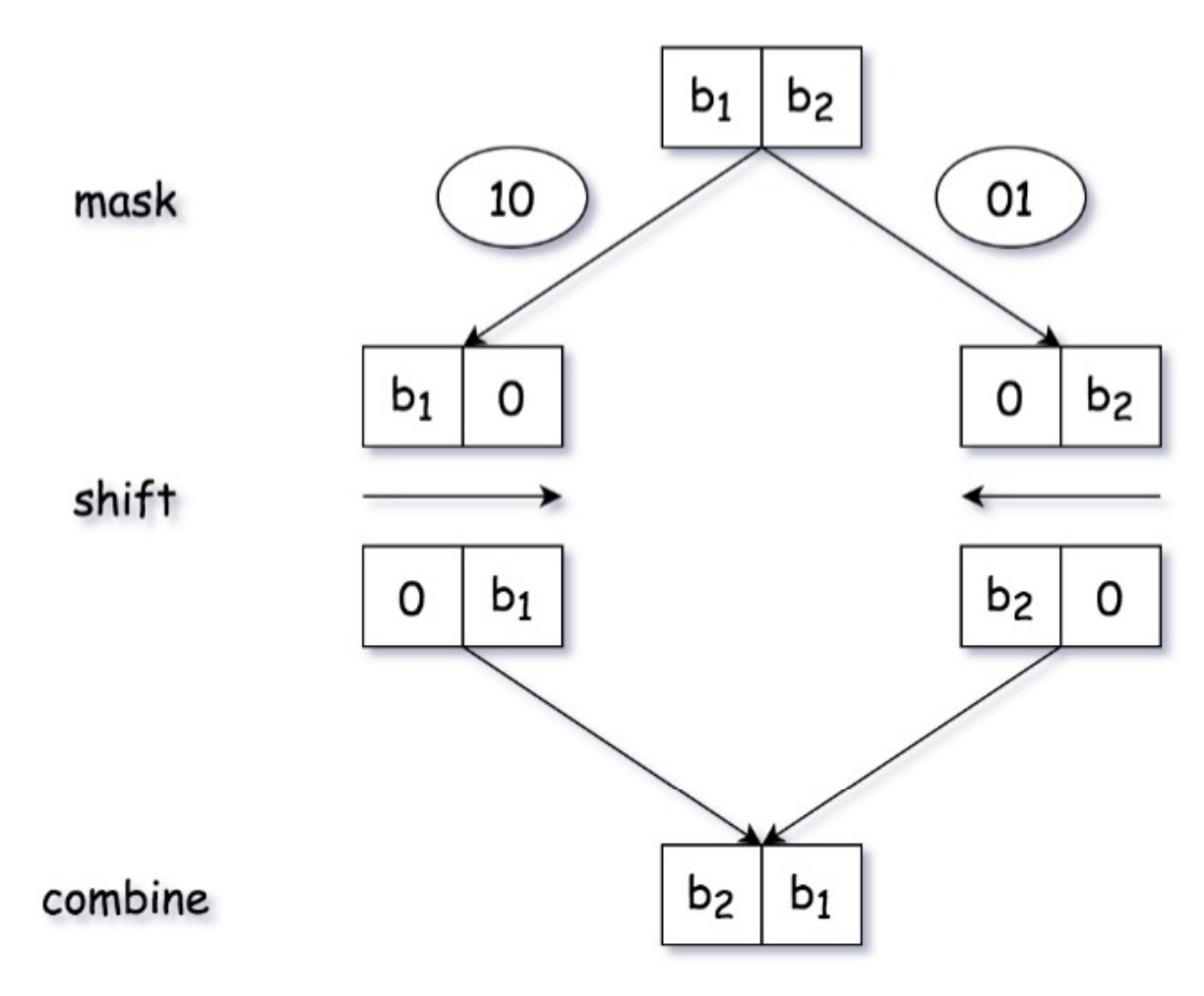
Approach 3: Mask and Shift

Intuition

We have shown in Approach #2 an example on how to reverse the bits in a byte without resorting to the loop statement. During the interview, one might be asked to reverse the entire 32 bits without using loop. Here we propose one solution that utilizes only the bit operations.

> The idea can be considered as a strategy of ***divide and conquer***, where we divide the original 32-bits into blocks with fewer bits via **bit masking**, then we reverse each block via **bit shifting**, and at the end we merge the result of each block to obtain the final result.

In the following graph, we demonstrate how to reverse two bits with the above-mentioned idea. As one can see, the idea could be applied to **blocks** of bits.



Algorithm

We can implement the algorithm in the following steps:

- 1). First, we break the original 32-bit into 2 blocks of 16 bits, and switch them.
- 2). We then break the 16-bits block into 2 blocks of 8 bits. Similarly, we switch the position of the 8-bits blocks
- 3). We then continue to break the blocks into smaller blocks, until we reach the level with the block of 1 bit.
- 4). At each of the above steps, we merge the intermediate results into a single integer which serves as the input for the next step.

The credit of this solution goes to @tworuler and @bhch3n for their post and comment in the discussion forum.

C++ | Python | Go — Copy

```cpp
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        n = (n >> 16) | (n << 16);
        n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8);
        n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4);
        n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2);
        n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1);
        return n;
    }
};
```

Complexity

- Time Complexity: $\mathcal{O}(1)$, no loop is used in the algorithm.
- Space Complexity: $\mathcal{O}(1)$. Actually, we did not even create any new variable in the function.

Rate this article: ★★★★★

Previous Next

Comments: 16

Sort By

Type comment here... (Markdown is supported)

Preview Post

jinsiang ★ 25 May 27, 2020 1:12 PM

It is a good question, and I don't think it is a easy level problem, although the code is short.

22 Share Reply

munkhbat ★ 22 May 22, 2020 10:59 PM

java version.

```
public class Solution {
    public int reverseBits(int n) {
```

Read More

10 Share Reply

Opimenov ★ 24 April 27, 2020 5:05 AM

to get the bit value bit AND has to be used. In your explanation you state that n | 1 will give you the value of the right most bit. It is incorrect. Please correct, so people don't learn the wrong way, as anything OR with 1 will always give you 1.

5 Share Reply

SHOW 1 REPLY

ShaneTsui ★ 109 April 17, 2020 9:00 AM

Mask & shift, but simpler.

```
class Solution:
    def reverseBits(self, n: int) -> int:
```

Read More

4 Share Reply

SHOW 1 REPLY

carlos6125 ★ 12 June 12, 2020 10:33 AM

Can't we say approach one is constant? Since we're always going to have 32 bits - O(32) = O(1).

1 Share Reply

SHOW 1 REPLY

dwbku ★ 1 June 8, 2020 10:56 PM

I don't understand the time complexity analysis, you have changed the assumption of whether the number is at most 32 bits in each approach and that's the only reason the complexity changes. Approach 1 is also O(1) given that the number is no more than 32 bits.

If you allow any number of bits then approach 2 is also O(log2N) as it is only faster by a constant factor,

Read More

1 Share Reply

awsmankitakalra ★ 2 May 1, 2020 6:52 AM

@liaison and @andvary

I think in explanation you should mention bitwise AND instead of bitwise OR.

1 Share Reply

SHOW 1 REPLY

darkpotter ★ 19 June 19, 2020 12:35 PM

Why its easy level ?

1 Share Reply

SHOW 1 REPLY

dyckia ★ 204 2 days ago

Please add java solution

0 Share Reply

gauravmvm11 ★ 0 July 5, 2020 9:44 PM

```
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
```

Read More

0 Share Reply

‹ 1 2 ›