190. Reverse Bits

✅ [JAVA / C++] : 0ms | O(1) Time

akshayaamar05 ★ 560

COMPLEXITY

- **Time: O(1)**, constant time
- **Space: O(1)**, in-place

BASIC IDEA

In this implementation we have followed **"Divide and Conquer"** strategy where **Original problem is divided into sub problems**

C++ | Java | Copy

```
public class Solution {

    public int reverseBits(int num) {

        num = ((num & 0xffff0000) >>> 16) | ((num & 0x0000ffff) << 16);
        num = ((num & 0xff00ff00) >>> 8) | ((num & 0x00ff00ff) << 8);
        num = ((num & 0xf0f0f0f0) >>> 4) | ((num & 0x0f0f0f0f) << 4);
        num = ((num & 0xcccccccc) >>> 2) | ((num & 0x33333333) << 2);
```

59

akshayaamar05 ★
560

COMPLEXITY

- **Time: O(1)**, constant time
- **Space: O(1)**, in-place

BASIC IDEA

In this implementation we have followed **"Divide and Conquer"** strategy where **Original problem is divided into sub problems**

C++ | Java | Copy

```java
public class Solution {

    public int reverseBits(int num) {

        num = ((num & 0xffff0000) >>> 16) | ((num & 0x0000ffff) << 16);
        num = ((num & 0xff00ff00) >>> 8) | ((num & 0x00ff00ff) << 8);
        num = ((num & 0xf0f0f0f0) >>> 4) | ((num & 0x0f0f0f0f) << 4);
        num = ((num & 0xcccccccc) >>> 2) | ((num & 0x33333333) << 2);
        num = ((num & 0xaaaaaaaa) >>> 1) | ((num & 0x55555555) << 1);

```

Let's understand in terms of decimal number to understand how the code is implemented
Suppose we have a number `12345678` and we have to reverse it to get `87654321` as desired output

The process will be as follows:

`12345678` --> original number

1. `56781234`
2. `78563412`
3. `87654321` --> desired number(reversed number)

Explanation of above process is as follows:

- Divide original number `(12345678)` into 2 parts(**4 - 4 each**)
  `1234|5678` and swap with each other i.e.
  |_____|

  `5678|1234`(it can also be said that we are **right shifting** the 1st part `(1234)` **to 4 places** from its original position and **left shifting** the 2nd part `(5678)` **to 4 places** from its original position)

- Divide this obtained number `(56781234)` into 4 parts(**2 - 2 each**)
  `56|78|12|34` and swap with each other i.e.
  |__| |__|

  `78|56|34|12`(it can also be said that we are **right shifting** the 1st part `(56)` and 3rd part `(12)` **to 2 places** from their original positions and **left shifting** the 2nd part `(78)` and 4th part `(34)` **to 2 places** from their original positions)

- Divide the obtained number `(78563412)` into 8 parts(**1 - 1 each**) `7|8|5|6|3|4|1|2` and swap with each other i.e.
  |_| |_| |_| |_|

  `8|7|6|5|4|3|2|1`(it can also be said that we are **right shifting** the 1st part `(7)`, 3rd part `(5)`, 5th part `(3)` and 7th part `(1)` **to 1 place** from their original positions and **left shifting** the 2nd part `(8)`, 4th part `(6)`, 6th part `(4)` and 8th part `(2)` **to 1 place** from their original positions)

  We got the desired output as `87654321`

Time to play with bits!!!!!!

To get better understanding of how the 32 bits are reversed in binary, we will take 8 bits instead of 32.
If the number is of 8 bits, the bits will be reversed in 3 steps as we are using **Divide and Conquer** approach which is nothing **dividing the original problem into sub problems** i.e. log(O(Number_Of_Bits)) i.e. log(O(8)) --> 3 and the same Idea applies **for 32 bits** where **the bits will be reversed in 5 steps** as log(O(32)) --> 5

First let's understand with 8 bits
Suppose we have bits as 00010111 and we have to reverse it to get 11101000 as desired output
The Process will be as follows:
00010111(8 bits) --> Original Number

1. 01110001
2. 11010100
3. 11101000 --> Reversed Numer

Explanation of above process is as follows:

1. Divide original bits into 4 - 4 each (4 * 2 = 8 bits)
   `0001|0111` and swap with each other i.e.
   |_____|
   `0111|0001` (It can also be said that we are **right shifting** 1st part(first 4 bits) **to 4 places** from their original positions and **left shifting** the 2nd part(last 4 bits) **to 4 places** from their original positions)

   Following is the process of doing it:
   a) **Preserve 1st part(first 4 bits)** and we know the property of bitwise and(&) opertor i.e. 0, 1 -> 0 and 1, 1 -> 1
   For this, we will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve the first 4 bits**
   mask = 0xf0 (which is nothing but `1111 0000` i.e. `1111` (15 == f) and 0000(0))
   `0001 0111` --> num
   & `1111 0000` --> 0xf0
   `0001 0000`

   b) **Right shift** the obtained number from its original position **by 4 places** i.e. (num & 0xf0) >>> 4
   `00000001`

c) **Preserve the 2nd part(last 4 bits)**
For this, will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve the last 4 bits**
mask = 0x0f (which is nothing but `0000 1111` i.e. `0000` (0) and `1111` (15 == f))
`0001 0111` --> num
& `0000 1111` --> 0x0f
`0000 0111`

d) **Left shift** the obtained number from its original position **by 4 places** i.e. (num & 0x0f) << 4
`01110000`

e) **Do the bitwise OR(|)** operation on both shifted numbers to **merge intermediate results** into a single number which is used as an input for the next step.
`0000 0001` --> number obtained by right shift at step b)
| `0111 0000` --> number obtained by left shift at step d)
`0111 0001`

f) **Assign the result into num** after apply bitwise or into num again to proceed furthur
num = `01110001`

Till here, 1 of 3 steps of process has been completed. 2 More remaining!!!

2. Divide obtained bits(`01110001`) into 2 - 2 each (2 * 4 = 8 bits)
   `01|11|00|01` and swap with each other i.e.
   |__| |__|
   `11|01|01|00` (It can also be said that we are **right shifting** 1st part(01) and 3rd part(00) **to 2 places** from their original positions and **left shifting** the 2nd part(11) and 4th part(01) **to 2 places** from their original positions)

   Following is the process of doing it:
   a) **Preserve 1st part(01) and 3rd part(00)** and we know the property of bitwise and(&) opertor i.e. 0, 1 -> 0 and 1, 1 -> 1
   For this, we will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve 1st part(01) and 3rd part(00)**
   mask = 0xcc (which is nothing but 1100 1100 i.e. (12 == c) and (12 == c))
   `01 11 00 01` --> num
   & `11 00 11 00` --> 0xcc
   `01 00 00 00`

   b) **Right shift** the obtained number(`01 00 00 00`) from its original position **by 2 places** i.e. (num & 0xcc) >>> 2
   `00 01 00 00`

c) **Preserve the 2nd part(11) and 4th part(01)**
For this, we will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve 2nd part(11) and 4th part(01)**
mask = 0x33 (which is nothing but 0011 0011 i.e. 0011(3) and 0011(3))

`01 11 00 01` --> num
& `00 11 00 11` --> 0x33
`00 11 00 01`

d) **Left shift** the obtained number(00 11 00 01) from its original position **by 2 places** i.e. (num & 0x33) << 2

`11 00 01 00`

e) **Do the bitwise OR(|)** operation on both shifted numbers to **merge intermediate results** into a single number which is used as an input for the next step.

`00 01 00 00` --> number obtained by right shift at step b)
| `11 00 01 00` --> number obtained by left shift at step d)
`11 01 01 00`

f) **Assign the result into num** after apply bitwise or into num again to proceed furthur

num = `11010100`

Till here, 2 of 3 steps of process has been completed. Only 1 more to go!!!!!!!!!

3. Divide obtained bits(`11010100`) into 1 - 1 each (1 * 8 = 8 bits)
   1|1|0|1|0|1|0|0 and swap with each other i.e.
   |_| |_| |_| |_|
   1|1|1|0|1|0|0|0 (It can also be said that we are **right shifting** 1st(1), 3rd(0), 5th(0) and 7th(0) parts **to 1 place** from their original positions and **left shifting** the 2nd(1), 4th(1), 6th(1) and 8th(0) parts **to 1 place** from their original positions)

   Following is the process of doing it
   a) **Preserve 1st(1), 3rd(0), 5th(0) and 7th(0) parts**
   We know the property of bitwise and(&) opertor i.e. 0, 1 -> 0 and 1, 1 -> 1
   For this, we will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve 1st(1), 3rd(0), 5th(0) and 7th(0) parts**
   mask = 0xaa (which is nothing but `1010 1010` i.e. (10 == a) and (10 == a))
   `1 1 0 1 0 1 0 0` --> num
   & `1 0 1 0 1 0 1 0` --> 0xaa
   `1 0 0 0 0 0 0 0`

   b) **Right shift** the obtained number(`1 0 0 0 0 0 0 0`) from its original position **by 1 place** i.e. (num & 0xaa) >>> 1
   `0 1 0 0 0 0 0 0`

c) **Preserve the 2nd(1), 4th(1), 6th(1) and 8th(0) parts**
For this, we will take a mask in **hexadecimal form** and **apply bitwise and(&) to preserve 2nd(1), 4th(1), 6th(1) and 8th(0) parts**
mask = 0x55 (which is nothing but 0101 0101 i.e. 0101(5) and 0101(5))

```
  1 1 0 1 0 1 0 0  --> num
& 0 1 0 1 0 1 0 1  --> 0x55
  0 1 0 1 0 1 0 0
```

d) **Left shift** the obtained number(0 1 0 1 0 1 0 0) from its original position **by 1 place** i.e. (num & 0x55) << 1

```
  1 0 1 0 1 0 0 0
```

e) **Do the bitwise OR(|)** operation on both shifted numbers

```
  0 1 0 0 0 0 0 0  --> number obtained by right shift at step b)
| 1 0 1 0 1 0 0 0  --> number obtained by left shift at step d)
  1 1 1 0 1 0 0 0
```

f) **Assign the result into num** after apply bitwise or into num again
num = `11101000`

Now, **return the num**.

We have finally reversed the original number i.e.

left shift at step d)

`1 1 1 0 1 0 0 0`

f) **Assign the result into num** after apply bitwise or into num again

num = `11101000`

Now, **return the num**.

We have finally reversed the original number i.e. `00010111` **->** `11101000`

Same idea goes for 32 bits
eg:
break the 32 bits into half(16 - 16 each) and right shift 1st half part to 16 positions and left shift the 2nd half to 16 positions
break the 16 bits into half(8 - 8 each) and right shift to 8 positions and left shift to 8 positions
break the 8 bits into half(4 - 4 each) and right shift to 4 positions and left shift to 4 positions
break the 4 bits into half(2 - 2 each) and right shift to 2 positions and left shift to 2 positions
break the 2 bits into half(1 - 1 each) and right shift to 1 positions and left shift to 1 positions

Refer to the following github repsitory for more leetcode solutions
https://github.com/Akshaya-Amar/LeetCodeSolutions