Bit Manipulation

Li Yin[1]

February 15, 2019

[1] www.liyinscience.com

Bit Manipulation

In this chapter, we will introduce basic knowledge about bit manipulation using Python. We found that a lot of popular Python book or even algorithm book they do not usually cover the topic of bit manipulation. However, mastering some basics operators, properties and knowing how bit manipulation sometimes can be applied to either save space or time complexity of your algorithm.

For example, how to convert a char or integer to bit, how to get each bit, set each bit, and clear each bit. Also, some more advanced bit manipulation operations. After this, we will see some examples to show how to apply bit manipulation in real-life problems.

0.1 Python Bitwise Operators

Bitwise operators include «, », &, |, ˜, ˆ. All of these operators operate on signed or unsigned numbers, but instead of treating that number as if it were a single value, they treat it as if it were a string of bits. Twos-complement binary is used for representing the singed number.

Now, we introduce the six bitwise operators.

x « y Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2^y.

x » y Returns x with the bits shifted to the right by y places. This is the same as dividing x by 2^y, same result as the // operator. This right shift is also called *arithmetic right shift*, it fills in the new bits with the value of the sign bit.

x & y "Bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0. It has the following property:

```
# keep 1 or 0 the same as original
1 & 1 = 1
```

```
0 & 1 = 0
# set to 0 with & 0
1 & 0 = 0
0 & 0 = 0
```

x | y "Bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

```
# set to 1 with | 1
1 | 1 = 1
0 | 1 = 1

# keep 1 or 0 the same as original
1 | 0 = 1
0 | 0 = 0
```

$\sim x$ Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as $-x-1$(really?).

x $\wedge$ **y** "Bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1. It has the following basic properties:

```
# toggle 1 or 0 with ^ 1
1 ^ 1 = 0
0 ^ 1 = 1

# keep 1 or 0 with ^ 0
1 ^ 0 = 1
0 ^ 0 = 0
```

Some examples shown:

```
A = 5 = 0101, B = 3 = 0011
 A ^ B = 0101 ^ 0011 = 0110 = 6

```

More advanced properties of XOR operator include:

```
a ^ b = c
c ^ b = a

n ^ n = 0
n ^ 0 = n
eg. a=00111011, b=10100000 , c= 10011011, c ^b= a

```

Logical right shift The logical right shift is different to the above right shift after shifting it puts a 0 in the most significant bit. It is indicated with a >>> operator n Java. However, in Python, there is no such operator, but we can implement one easily using **bitstring** module padding with zeros using >>= operator.

```
>>> a = BitArray(int=-1000, length=32)
>>> a.int
-1000
>>> a >>= 3
>>> a.int
536870787
```

0.2 Python Built-in Functions

bin() The bin() method takes a single parameter **num-** an integer and return its *binary string*. If not an integer, it raises a TypeError exception.

```
a = bin(88)
print(a)
# output
# 0b1011000
```

However, bin() doesn't return *binary bits* that applies the two's complement rule. For example, for the negative value:

```
a1 = bin(-88)
# output
# -0b1011000
```

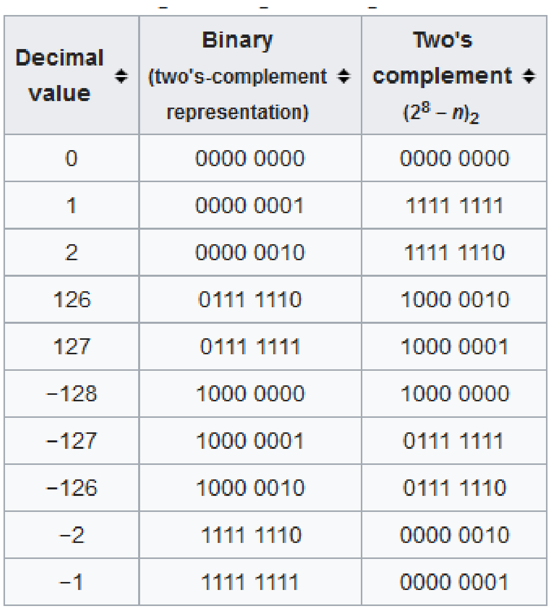
int(x, base = 10) The int() method takes either a string x to return an integer with its corresponding base. The common base are: 2, 10, 16 (hex).

```
b = int('01011000', 2)
c = int('88', 10)
print(b, c)
# output
# 88 88
```

chr() The chr() method takes a single parameter of integer and return a character (a string) whose Unicode code point is the integer. If the integer i is outside the range, ValueError will be raised.

```
d = chr(88)
print(d)
# output
# X
```

ord() The ord() method takes a string representing one Unicode character and return an integer representing the Unicode code point of that character.



Decimal value	Binary (two's-complement representation)	Two's complement $(2^8 - n)_2$
0	0000 0000	0000 0000
1	0000 0001	1111 1111
2	0000 0010	1111 1110
126	0111 1110	1000 0010
127	0111 1111	1000 0001
−128	1000 0000	1000 0000
−127	1000 0001	0111 1111
−126	1000 0010	0111 1110
−2	1111 1110	0000 0010
−1	1111 1111	0000 0001

Figure 1: Two's Complement Binary for Eight-bit Signed Integers.

```python
e = ord('a')
print(e)
# output
#  97
```

0.3 Twos-complement Binary

Given 8 bits, if it is unsigned, it can represent the values 0 to 255 (1111,1111). However, a two's complement 8-bit number can only represent positive integers from 0 to 127 (0111,1111) because the most significant bit is used as sign bit: '0' for positive, and '1' for negative.

$$\sum_{i=0}^{N-1} 2^i = 2^{(N-1)} + 2^{(N-2)} + ... + 2^2 + 2^1 + 2^0 = 2^N - 1 \tag{1}$$

The twos-complement binary is the same as the classical binary representation for positive integers and differs slightly for negative integers. Negative integers are represented by performing Two's complement operation on its absolute value: it would be $(2^N - n)$ for representing $-n$ with N-bits. Here, we show Two's complement binary for eight-bit signed integers in Fig. 1.

Get Two's Complement Binary Representation In Python, to get the two's complement binary representation of a given integer, we do not really have a built-in function to do it directly for negative number. Therefore, if we want to know how the two's complement binary look like for negative integer we need to write code ourselves. The Python code is given as:

```
bits = 8
ans = (1 << bits) -2
print(ans)
# output
# '0b11111110'
```

There is another method to compute: inverting the bits of n (this is called **One's Complement**) and adding 1. For instance, use 8 bits integer 5, we compute it as the follows:

$$5_{10} = 0000,0101_2, \tag{2}$$
$$-5_{10} = 1111,1010_2 + 1_2, \tag{3}$$
$$-5_{10} = 1111,1011_2 \tag{4}$$

To flip a binary representation, we need expression x XOR '1111,1111', which is $2^N - 1$. The Python Code is given:

```
def twos_complement(val, bits):
    # first flip implemented with xor of val with all 1's
    flip_val = val ^ (1 << bits - 1)
    #flip_val = ~val we only give 3 bits
    return bin(flip_val + 1)
```

Get Two's Complement Binary Result In Python, if we do not want to see its binary representation but just the result of two's complement of a given positive or negative integer, we can use two operations $-x$ or $\sim +1$. For input 2, the output just be a negative integer -2 instead of its binary representation:

```
def twos_complement_result(x):
    ans1 = -x
    ans2 = ~x + 1
    print(ans1, ans2)
    print(bin(ans1), bin(ans2))
    return ans1
# output
# -8 -8
# -0b1000 -0b1000
```

This is helpful if we just need two's complement result instead of getting the binary representation.

0.4 Useful Combined Bit Operations

For operations that handle each bit, we first need a *mask* that only set that bit to 1 and all the others to 0, this can be implemented with arithmetic left shift sign by shifting 1 with 0 to n-1 steps for n bits:

```
mask = 1 << i
```

Get ith Bit In order to do this, we use the property of AND operator either 0 or 1 and with 1, the output is the same as original, while if it is and with 0, they others are set with 0s.

```
# for n bit, i in range [0,n-1]
def get_bit(x, i):
    mask = 1 << i
    if x & mask:
        return 1
    return 0
print(get_bit(5,1))
# output
# 0
```

Else, we can use left shift by i on x, and use AND with a single 1.

```
def get_bit2(x, i):
    return x >> i & 1
print(get_bit2(5,1))
# output
# 0
```

Set ith Bit We either need to set it to 1 or 0. To set this bit to 1, we need matching relation: $1->1, 0->1$. Therefore, we use operator |. To set it to 0: $1->0, 0->0$. Because 0 & 0/1 = 0, 1&0=1, 1&1 = 1, so we need first set that bit to 0, and others to 1.

```
# set it to 1
x = x | mask

# set it to 0
x = x & (~mask)
```

Toggle ith Bit Toggling means to turn bit to 1 if it was 0 and to turn it to 0 if it was one. We will be using 'XOR' operator here due to its properties.

```
x = x ^ mask
```

Clear Bits In some cases, we need to clear a range of bits and set them to 0, our base mask need to put 1s at all those positions, Before we solve this problem, we need to know a property of binary subtraction. Check if you can find out the property in the examples below,

```
1000-0001 = 0111
0100-0001 = 0011
1100-0001 = 1011
```

The property is, the difference between a binary number n and 1 is all the bits on the right of the rightmost 1 are flipped including the rightmost 1. Using this amazing property, we can create our mask as:

```
# base mask
i = 5
mask = 1 << i
mask = mask -1
print(bin(mask))
# output
# 0b11111
```

With this base mask, we can clear bits: (1) All bits from the most significant bit till i (leftmost till ith bit) by using the above mask. (2) All bits from the lest significant bit to the ith bit by using $\sim mask$ as mask. The Python code is as follows:

```
# i i-1 i-2 ... 2 1 0, keep these positions
def clear_bits_left_right(val, i):
    print('val', bin(val))
    mask = (1 << i) -1
    print('mask', bin(mask))
    return bin(val & (mask))
```

```
# i i-1 i-2 ... 2 1 0, erase these positions
def clear_bits_right_left(val, i):
    print('val', bin(val))
    mask = (1 << i) -1
    print('mask', bin(~mask))
    return bin(val & (~mask))
```

Run one example:

```
print(clear_bits_left_right(int('11111111',2), 5))
print(clear_bits_right_left(int('11111111',2), 5))
val 0b11111111
mask 0b11111
0b11111
val 0b11111111
mask -0b100000
0b11100000
```

Get the lowest set bit Suppose we are given '0010,1100', we need to get the lowest set bit and return '0000,0100'. And for 1100, we get 0100. If we try to do an AND between 5 and its two's complement as shown in Eq. 2 and 4, we would see only the right most 1 bit is kept and all the others are cleared to 0. This can be done using expression $x\&(-x)$, $-x$ is the two's complement of x.

```
def get_lowest_set_bit(val):
    return bin(val & (-val))
print(get_lowest_set_bit(5))
# output
# 0b1
```

Or, optionally we can use the property of subtracting by 1.

```
x ^ (x -1))
```

Clear the lowest set bit In many situations we want to strip off the lowest set bit for example in Binary Indexed tree data structure, counting number of set bit in a number. We use the following operations:

```python
def strip_last_set_bit(val):
    print(bin(val))
    return bin(val & (val - 1))
print(strip_last_set_bit(5))
# output
# 0b101
# 0b100
```

0.5 Applications

Recording States Some algorithms like Combination, Permutation, Graph Traversal require us to record states of the input array. Instead of using an array of the same size, we can use a single integer, each bit's location indicates the state of one element with same index in the array. For example, we want to record the state of an array with length 8. We can do it like follows:

```python
used = 0
for i in range(8):
    if used &(1<<i): # check state at i
        continue
    used = used | (1<<i)  # set state at i used
    print(bin(used))
```

It has the following output

```
0b1
0b11
0b111
0b1111
0b11111
0b111111
0b1111111
0b11111111
```

XOR Single Number

0.1 **136. Single Number(easy).** Given a non-empty array of integers, every element appears twice except for one. Find that single one. *Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?*

```
Example 1:

Input: [2,2,1]
Output: 1

Example 2:

Input: [4,1,2,1,2]
Output: 4
```

Solution: XOR. This one is kinda straightforward. You'll need to know the properties of XOR as shown in Section 0.1.

```
n ^ n = 0
n ^ 0 = n
```

Therefore, we only need on variable to record the state which is initialize with 0: the first time to appear x = n, second time to appear x = 0. the last element x will be the single number. To set the statem we can use XOR.

```python
def singleNumber(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    v = 0
    for e in nums:
        v = v ^ e
    return v
```

0.2 **137. Single Number II** Given a non-empty array of integers, every element appears three times except for one, which appears exactly once. Find that single one. *Note: Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?*

```
Example 1:

Input: [2,2,3,2]
Output: 3

Example 2:

Input: [0,1,0,1,0,1,99]
Output: 99
```

Solution: XOR and Two Variables. In this problem, because all element but one appears three times. To record the states of three, we need at least two variables. And we initialize it to a = 0, b = 0. For example, when 2 appears the first time, we set a = 2, b = 0; when it appears two times, a = 0, b = 2; when it appears three times, a = 0, b

= 0. For number that appears one or two times will be saves either in a or in b. Same as the above example, we need to use XOR to change the state for each variable. We first do a = a XOR v, b = XOR v, we need to keep a unchanged and set b to zero. We can do this as a = a XOR v & ∼ b; b = b XOR v & ∼ a.

```
def singleNumber(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    a = b = 0
    for num in nums:
        a = a ^ num & ~b
        b = b ^ num & ~a
    return a|b
```

0.3 421. Maximum XOR of Two Numbers in an Array (medium).

Given a non-empty array of numbers, $a_0, a_1, a_2, ..., a_{n-1}$, where $0 \leq a_i < 2^{31}$. Find the maximum result of a_i XOR a_j, where $0 \leq i, j < n$. Could you do this in $O(n)$ runtime?

```
Example:
Input: [3, 10, 5, 25, 2, 8]

Output: 28
Explanation: The maximum result is 5 \^ 25 = 28.
```

Solution 1: Build the Max bit by bit. First, let's convert these integers into binary representation by hand.

```
3    0000, 0011
10   0000, 1011
5    0000, 0101
25   0001, 1001
2    0000, 0010
8    0000, 1000
```

If we only look at the highest position i where there is one one and all others zero. Then we know the maximum XOR m has 1 at that bit. Now, we look at two bits: i, i-1. The possible maximum XOR for this is append 0 or 1 at the end of m, we have possible max 11, because for XOR, if we do XOR of m with others, $mXORa = b$, if b exists in these possible two sets, then max is possible and it become $m << 1 + 1$. We can carry on this process, the following process is showed as follows: answer$\hat{1}$ is the possible max,

```
def findMaximumXOR(self, nums):
    """
    :type nums: List[int]
    :rtype: int
    """
    answer = 0
    for i in range(32)[::-1]:
        answer <<= 1 # multiple it by two
        prefixes = {num >> i for num in nums} # shift right
 for n, divide/2^i, get the first (32-i) bits
        answer += any((answer+1) ^ p in prefixes for p in
    prefixes)
    return answer
```

Solution 2: Use Trie.

```python
def findMaximumXOR(self, nums):
    def Trie():
        return collections.defaultdict(Trie)

    root = Trie()
    best = 0

    for num in nums:
        candidate = 0
        cur = this = root
        for i in range(32)[::-1]:
            curBit = num >> i & 1
            this = this[curBit]
            if curBit ^ 1 in cur:
                candidate += 1 << i
                cur = cur[curBit ^ 1]
            else:
                cur = cur[curBit]
        best = max(candidate, best)
    return best
```

With Mask

0.4 **190. Reverse Bits (Easy).**Reverse bits of a given 32 bits unsigned integer.

```
Example 1:

Input: 00000010100101000001111010011100
Output: 00111001011110000010100101000000
Explanation: The input binary string
    00000010100101000001111010011100 represents the unsigned
     integer 43261596, so return 964176192 which its binary
    representation is 00111001011110000010100101000000.

Example 2:

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

```
 integer 4294967293, so return 3221225471 which its
binary representation is
10101111110010110010011101101001.
```

Solution: Get Bit and Set bit with mask. We first get bits from the most significant position to the least significant position. And get the bit at that position with mask, and set the bit in our 'ans' with a mask indicates the position of (31-i):

```
# @param n, an integer
# @return an integer
def reverseBits(self, n):
    ans = 0
    for i in range(32)[::-1]: #from high to low
        mask = 1 << i
        set_mask = 1 << (31-i)
        if (mask & n) != 0: #get bit
            #set bit
            ans |= set_mask
    return ans
```

0.5 201. Bitwise AND of Numbers Range (medium).

Given a range [m, n] where $0 \leq m \leq n \leq 2147483647$, return the bitwise AND of all numbers in this range, inclusive.

```
Example 1:

Input: [5,7]
Output: 4

Example 2:

Input: [0,1]
Output: 0
```

Solution 1: O(n) do AND operation. We start a 32 bit long 1s. The solution would receive LTE error.

```
def rangeBitwiseAnd(self, m, n):
    """
    :type m: int
    :type n: int
    :rtype: int
    """
    ans = int('1'*32, 2)
    for c in range(m, n+1):
        ans &= c
    return ans
```

Solution 2: Use mask, check bit by bit. Think, if we AND all, the resulting integer would definitely smaller or equal to m. For example 1:

```
0101 5
0110 6
0111 7
```

We start from the least significant bit at 5, if it is 1, then we check the closest number to 5 that has 0 at the this bit. It would be 0110. If this number is in the range, then this bit is offset to 0. We then move on to check the second bit. To make this closest number: first we clear the least i+1 positions in m to get 0100 and then we add it with $1 << (i + 1)$ as 0010 to get 0110.

```python
def rangeBitwiseAnd(self, m, n):
    ans = 0
    mask = 1
    for i in range(32): # [::-1]:
        bit = mask & m != 0
        if bit:
            # clear i+1, ..., 0
            mask_clear = (mask<<1)-1
            left = m & (~mask_clear)
            check_num = (mask << 1) + left
            if check_num < m or check_num > n:
                ans |= 1 << i
        mask = mask << 1
    return ans
```

Solution 3: Use While Loop. We can start do AND of n with (n-1). If the resulting integer is still larger than m, then we keep do such AND operation.

```python
def rangeBitwiseAnd(self, m, n):
    ans=n
    while ans>m:
        ans=ans&(ans-1)
    return ans
```

0.6 LeetCode Problems

1. Write a function to determine the number of bits required to convert integer A to integer B.

```python
def bitswaprequired(a, b):
  count = 0
  c = a^b
  while(c != 0):
    count += c & 1
    c = c >> 1
  return count
print(bitswaprequired(12, 7))
```

2. **389. Find the Difference (easy).** Given two strings s and t which consist of only lowercase letters. String t is generated by random shuffling string s and then add one more letter at a random position. Find the letter that was added in t.

```
Example:
Input:
s = "abcd"
t = "abcde"

Output:
e
Explanation:
'e' is the letter that was added.
```

   Solution 1: Use Counter Difference. This way we need $O(M+N)$ space to save the result of counter for each letter.

```python
def findTheDifference(self, s, t):
    s = collections.Counter(s)
    t = collections.Counter(t)
    diff = t - s
    return list(diff.keys())[0]
```

   Solution 2: Single Number with XOR. Using bit manipulation and with $O(1)$ we can find it in $O(M+N)$ time, which is the best BCR:

```python
def findTheDifference(self, s, t):
    """
    :type s: str
    :type t: str
    :rtype: str
    """
    v = 0
    for c in s:
        v = v ^ ord(c)
    for c in t:
        v = v ^ ord(c)
    return chr(v)
```

3. **50. Pow(x, n) (medium).** for n, such as 10, we represent it as 1010, if we have a base and an result, we start from the least significant position, each time we move, the base because base*base, and if the value if 1, then we multiple the answer with the base.