Sum of Two Integers (/category/455/sum-of-two-integers)
/ A summary: how to use bit manipulation to solve problems easily and efficiently (/to…

New users please **read the instructions** (https://discuss.leetcode.com/topic/22/welcome-new-users-please-read-this-before-posting) to *format your code* properly.

Discuss is a place to **post interview questions** (/category/5/interview-questions) or share solutions / ask questions related to OJ problems (https://leetcode.com/problems).

Contribute and Win a LeetCode T-Shirt! (https://discuss.leetcode.com/category/767/operating-system)

Do you want to win a **LeetCode T-Shirt (https://leetcode.com/static/images/tshirt_promo.png)**? Do you have an interesting Operating System related interview question?

To contribute, submit the **question description**, **where have you first heard of this question**, and **a brief explanation of your solution** to the **Operating System (https://discuss.leetcode.com/category/767/operating-system)** subcategory.

We will select Top 10 questions as winners. To qualify winning, please submit by this weekend (**June 04, 2017**). Each user can win at most one T-Shirt.

417

(/user/lhearen) **LHearen (/user/lhearen)** **(/groups/global-moderators)**
Reputation: ★ 1,146

WIKI

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization. For most other tasks, modern programming languages allow the programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.

Bit manipulation, in some cases, can obviate or reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are processed in parallel, but the code can become more difficult to write and maintain.

DETAILS

Basics

At the heart of bit manipulation are the bit-wise operators & (and), | (or), ~ (not) and ^ (exclusive-or, xor) and shift operators a << b and a >> b.

> *There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a caret, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR.*

- Set union A | B
- Set intersection A & B
- Set subtraction A & ~B
- Set negation ALL_BITS ^ A or ~A
- Set bit A |= 1 << bit
- Clear bit A &= ~(1 << bit)
- Test bit (A & 1 << bit) != 0
- Extract last bit A&-A or A&~(A-1) or x^(x&(x-1))
- Remove last bit A&(A-1)
- Get all 1-bits ~0

Examples

Count the number of ones in the binary representation of the given number

```
int count_one(int n) {
    while(n) {
        n = n&(n-1);
        count++;
    }
    return count;
}
```

Is power of four (actually map-checking, iterative and recursive methods can do the same)

```
bool isPowerOfFour(int n) {
    return !(n&(n-1)) && (n&0x55555555);
    //check the 1-bit location;
}
```

^ tricks

Use `^` to remove even exactly same numbers and save the odd, or save the distinct bits and remove the same.

SUM OF TWO INTEGERS

Use `^` and `&` to add two integers

```
int getSum(int a, int b) {
    return b==0? a:getSum(a^b, (a&b)<<1); //be careful about the terminating condi
}
```

MISSING NUMBER

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array. For example, Given nums = [0, 1, 3] return 2. (Of course, you can do this by math.)

```
int missingNumber(vector<int>& nums) {
    int ret = 0;
    for(int i = 0; i < nums.size(); ++i) {
        ret ^= i;
        ret ^= nums[i];
    }
    return ret^=nums.size();
}
```

| tricks

Keep as many 1-bits as possible

Find the largest power of 2 (most significant bit in binary form), which is less than or equal to the given number N.

```
long largest_power(long N) {
    //changing all right side bits to 1.
    N = N | (N>>1);
    N = N | (N>>2);
    N = N | (N>>4);
    N = N | (N>>8);
    N = N | (N>>16);
    return (N+1)>>1;
}
```

REVERSE BITS

Reverse bits of a given 32 bits unsigned integer.

Solution

```
uint32_t reverseBits(uint32_t n) {
    unsigned int mask = 1<<31, res = 0;
    for(int i = 0; i < 32; ++i) {
        if(n & 1) res |= mask;
        mask >>= 1;
        n >>= 1;
    }
    return res;
}
```

```
uint32_t reverseBits(uint32_t n) {
        uint32_t mask = 1, ret = 0;
        for(int i = 0; i < 32; ++i){
                ret <<= 1;
                if(mask & n) ret |= 1;
                mask <<= 1;
        }
        return ret;
}
```

& tricks

Just selecting certain bits

Reversing the bits in integer

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

BITWISE AND OF NUMBERS RANGE

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive. For example, given the range [5, 7], you should return 4.

Solution

```
int rangeBitwiseAnd(int m, int n) {
    int a = 0;
    while(m != n) {
        m >>= 1;
        n >>= 1;
        a++;
    }
    return m<<a;
}
```

NUMBER OF 1 BITS

Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the Hamming weight).

Solution

```
int hammingWeight(uint32_t n) {
        int count = 0;
        while(n) {
                n = n&(n-1);
                count++;
        }
        return count;
}
```

```
int hammingWeight(uint32_t n) {
    ulong mask = 1;
    int count = 0;
    for(int i = 0; i < 32; ++i){ //31 will not do, delicate;
        if(mask & n) count++;
        mask <<= 1;
    }
    return count;
}
```

Application

REPEATED DNA SEQUENCES

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA. Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.
For example,
Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",
Return: ["AAAAACCCCC", "CCCCCAAAAA"].

Solution

```
class Solution {
public:
    vector<string> findRepeatedDnaSequences(string s) {
        int sLen = s.length();
        vector<string> v;
        if(sLen < 11) return v;
        char keyMap[1<<21]{0};
        int hashKey = 0;
        for(int i = 0; i < 9; ++i) hashKey = (hashKey<<2) | (s[i]-'A'+1)%5;
        for(int i = 9; i < sLen; ++i) {
            if(keyMap[hashKey = ((hashKey<<2)|(s[i]-'A'+1)%5)&0xfffff]++ == 1)
                v.push_back(s.substr(i-9, 10));
        }
        return v;
    }
};
```

> *But the above solution can be invalid when repeated sequence appears too many times, in which case we should use `unordered_map<int, int> keyMap` to replace `char keyMap[1<<21]{0}` here.*

MAJORITY ELEMENT

Given an array of size n, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times. (bit-counting as a usual way, but here we actually also can adopt sorting and Moore Voting Algorithm)

Solution

```
int majorityElement(vector<int>& nums) {
    int len = sizeof(int)*8, size = nums.size();
    int count = 0, mask = 1, ret = 0;
    for(int i = 0; i < len; ++i) {
        count = 0;
        for(int j = 0; j < size; ++j)
            if(mask & nums[j]) count++;
        if(count > size/2) ret |= mask;
        mask <<= 1;
    }
    return ret;
}
```

SINGLE NUMBER III

Given an array of integers, every element appears three times except for one. Find that single one. (Still this type can be solved by bit-counting easily.) But we are going to solve it by `digital logic design`

Solution

```
//inspired by logical circuit design and boolean algebra;
//counter - unit of 3;
//current   incoming  next
//a b           c    a b
//0 0           0    0 0
//0 1           0    0 1
//1 0           0    1 0
//0 0           1    0 1
//0 1           1    1 0
//1 0           1    0 0
//a = a&~b&~c + ~a&b&c;
//b = ~a&b&~c + ~a&~b&c;
//return a|b since the single number can appear once or twice;
int singleNumber(vector<int>& nums) {
    int t = 0, a = 0, b = 0;
    for(int i = 0; i < nums.size(); ++i) {
        t = (a&~b&~nums[i]) | (~a&b&nums[i]);
        b = (~a&b&~nums[i]) | (~a&~b&nums[i]);
        a = t;
    }
    return a | b;
}
;
```

MAXIMUM PRODUCT OF WORD LENGTHS

Given a string array words, find the maximum value of length(word[i]) * length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

> *Example 1:*
> *Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]*
> *Return 16*
> *The two words can be "abcw", "xtfn".*

> *Example 2:*
> *Given ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]*
> *Return 4*
> *The two words can be "ab", "cd".*

> *Example 3:*
> *Given ["a", "aa", "aaa", "aaaa"]*
> *Return 0*
> *No such pair of words.*

Solution

Since we are going to use the length of the word very frequently and we are to compare the letters of two words checking whether they have some letters in common:

- using an array of int to pre-store the length of each word reducing the frequently *measuring* process;
- since int has 4 bytes, a 32-bit type, and there are only 26 different letters, so we can just use one bit to indicate the existence of the letter in a word.

```
int maxProduct(vector<string>& words) {
    vector<int> mask(words.size());
    vector<int> lens(words.size());
    for(int i = 0; i < words.size(); ++i) lens[i] = words[i].length();
    int result = 0;
    for (int i=0; i<words.size(); ++i) {
        for (char c : words[i])
            mask[i] |= 1 << (c - 'a');
        for (int j=0; j<i; ++j)
            if (!(mask[i] & mask[j]))
                result = max(result, lens[i]*lens[j]);
    }
    return result;
}
```

Attention

- result after shifting left(or right) too much is undefined
- right shifting operations on negative values are undefined
- right operand in shifting should be non-negative, otherwise the result is undefined
- The & and | operators have lower precedence than comparison operators

SETS

All the subsets
A big advantage of bit manipulation is that it is trivial to iterate over all the subsets of an N-element set: every N-bit value represents some subset. Even better, `if A is a subset of B then the number representing A is less than that representing B`, which is convenient for some dynamic programming solutions.

It is also possible to iterate over all the subsets of a particular subset (represented by a bit pattern), provided that you don't mind visiting them in reverse order (if this is problematic, put them in a list as they're generated, then walk the list backwards). The trick is similar to that for finding the lowest bit in a number. If we subtract 1 from a subset, then the lowest set element is cleared, and every lower element is set. However, we only want to set those lower elements that are in the superset. So the iteration step is just `i = (i - 1) & superset`.

```
vector<vector<int>> subsets(vector<int>& nums) {
    vector<vector<int>> vv;
    int size = nums.size();
    if(size == 0) return vv;
    int num = 1 << size;
    vv.resize(num);
    for(int i = 0; i < num; ++i) {
        for(int j = 0; j < size; ++j)
            if((1<<j) & i) vv[i].push_back(nums[j]);
    }
    return vv;
}
```

Actually there are two more methods to handle this using `recursion` and `iteration` respectively.

BITSET

A bitset (http://www.cplusplus.com/reference/bitset/bitset/?kw=bitset) stores bits (elements with only two possible values: 0 or 1, true or false, ...).
The class emulates an array of bool elements, but optimized for space allocation: generally, each element occupies only one bit (which, on most systems, is eight times less than the smallest elemental type: char).

```
// bitset::count
#include <iostream>       // std::cout
#include <string>         // std::string
#include <bitset>         // std::bitset

int main () {
  std::bitset<8> foo (std::string("10110011"));
  std::cout << foo << " has ";
  std::cout << foo.count() << " ones and ";
  std::cout << (foo.size()-foo.count()) << " zeros.\n";
  return 0;
}
```

Always welcom new ideas and `practical` tricks, just leave them in the comments!

Jul 6, 2016, 2:33 PM (/post/111971) reply quote

J M K F F ›

Reply (https://leetcode.com/problems/sum-of-two-integers)

17

S (/user/sand) **SanD (/user/sand)**
Reputation: ★ 67

Because I suck at Bit manipulations, this is the most useful post in Leetcode for me. Thanks!

Jul 14, 2016, 4:38 AM (/post/113798) reply quote

2

M (/user/mengduo2) **mengduo2 (/user/mengduo2)**
Reputation: ★ 2

Thanks! The explanation for ^ and & is very thorough!

said in A summary: how to use bit manipulation to solve problems easily and efficiently (/post/null):

> *Use ^ to remove even exactly same numbers and save the odd, or save the distinct bits and remove the same.*

Aug 25, 2016, 10:09 AM (/post/124157) reply quote

2

M (/user/mikealive) **mikealive (/user/mikealive)**
Reputation: ★ 4

Thank you! Your summary is really good~

Sep 16, 2016, 3:43 AM (/post/131052) reply quote

1

C (/user/ceclinux) **ceclinux (/user/ceclinux)**
Reputation: ★ 2

Thank you very much for this summary, very helpful.

Sep 20, 2016, 5:55 PM (/post/132515) reply quote

1

C (/user/christopherpagan) **ChristopherPagan (/user/christopherpagan)**
Reputation: ★ 1

This is a very good post! Thank you! I'm not sure why it never occurred to me to use bit manipulation for the problem, although I am familiar with it. I'd like to suggest to include a piece on the difference between arithmetic/signed shift (https://en.wikipedia.org/wiki/Arithmetic_shift), the logical shift (https://en.wikipedia.org/wiki/Logical_shift) and the >>> operator in Java and other languages.

Sep 23, 2016, 10:59 PM (/post/133383) reply quote

1

W (/user/whysoserious) **whysoserious (/user/whysoserious)**
Reputation: ★ 2

@LHearen (https://discuss.leetcode.com/uid/1527) I think '^' operator will give us the sum of two numbers without the carry and '&' operator will give us the carry?

Nov 11, 2016, 3:45 PM (/post/147096) reply quote

1

(/user/lhearen) **LHearen (/user/lhearen)** (/groups/global-moderators)
Reputation: ★ 1,146

@whysoserious (https://discuss.leetcode.com/uid/6840) Yes, if you would like to treat all bits as the `number`, it's okay to think of it like that. It's then kind of different preferences. The best should always it that suits your thoughts.

Nov 11, 2016, 4:12 PM (/post/147103) reply quote

1

(/user/lhearen) **LHearen (/user/lhearen)** (/groups/global-moderators)
Reputation: ★ 1,146

@syrhuang (https://discuss.leetcode.com/uid/74724) Actually you will find it simple, if you do try it yourself in `bit` way.
Take a small example here, 6, whose bit format will be `110` then `6-1` will be `101` from the last 1-bit (the second 1-bit) till the last bit 0 all bits are reversed (10 -> 01) and then if we do `bit and &` here with itself only the former part (except the last valid 1-bit) will be reserved and then we can count 1-bit, since here we eliminate the last 1-bit. The left will be 4 - `100` and same logic - `4-1 -> 011` and we do `bit and &` with itself `100 & 011` -> zero. So all valid 1-bits are counted as expected.

Dec 3, 2016, 9:20 PM (/post/153153) reply quote

S

1

K (/user/krishnacs) **krishnacs (/user/krishnacs)**
Reputation: ★ 12

@LHearen (https://discuss.leetcode.com/uid/1527) said in A summary: how to use bit manipulation to solve problems easily and efficiently (/post/111971):

> *Repeated DNA Sequences*

Thank you very much!

Could you please elaborate tad more on "REPEATED DNA SEQUENCES" bit manipulation, please!

Mar 21, 2017, 9:51 AM (/post/178320) reply quote

0

Z (/user/zhiyuan6) **zhiyuan6 (/user/zhiyuan6)**
Reputation: ★ 0

Thank you for your summary, that's helpful!

Sep 26, 2016, 2:56 AM (/post/133938) reply quote

0

S (/user/soulsolja) **soulsolja (/user/soulsolja)**
Reputation: ★ 0

In the second version of hammingWeight, shouldn't you be shifting right not left?

Sep 28, 2016, 1:00 AM (/post/134474) reply quote

0

(/user/lhearen) **LHearen (/user/lhearen)** **(/groups/global-moderators)**
Reputation: ★ 1,146

@soulsolja (https://discuss.leetcode.com/uid/61907) You mean `mask <<= 1` to `mask >>= 1` ?

> *1 >>= 1 will be `zero` while 1 <<= 1 will be `two`*

Sep 29, 2016, 9:50 AM (/post/134903) reply quote

0

A (/user/ashish56) **ashish56 (/user/ashish56)**
Reputation: ★ 1

thank you very much :)

Oct 10, 2016, 11:46 PM (/post/137956) reply quote

0

W (/user/whysoserious) **whysoserious (/user/whysoserious)**
Reputation: ★ 2

What are your thought process of coming up the solution for Sum of two integers?

Nov 10, 2016, 1:53 PM (/post/146806) reply quote

0

(/user/lhearen) **LHearen (/user/lhearen)** **(/groups/global-moderators)**
Reputation: ★ 1,146

@whysoserious (https://discuss.leetcode.com/uid/6840) There are restrictions using `other operators` and instinctively I thought of the `bit` while `^` will collect the different and `&` collect the same and as a result I can retain the information I need to retrieve the final result.

Nov 11, 2016, 3:12 PM (/post/147084) reply quote

w ›

0

V (/user/viviannn) **viviannn (/user/viviannn)**
Reputation: ★ -5

Thanks for your sharing!

Nov 27, 2016, 10:30 AM (/post/151586) reply quote

0

S (/user/syrhuang) **syrhuang (/user/syrhuang)**
Reputation: ★ 0

about the count_one program, how do you come up with the idea n = n&(n-1) ? can you explain more? Thank you very much!

Dec 2, 2016, 12:43 PM (/post/152939) reply quote

0

S (/user/syrhuang) **syrhuang (/user/syrhuang)**
Reputation: ★ 0

@LHearen (https://discuss.leetcode.com/uid/1527) thank you for your explanation!

Dec 4, 2016, 7:01 AM (/post/153214) reply quote

0

C (/user/christoph-d) **Christoph-D (/user/christoph-d)**
Reputation: ★ 1

@LHearen (https://discuss.leetcode.com/uid/1527) said in A summary: how to use bit manipulation to solve problems easily and efficiently (/post/111971):

> *There is no boolean operator counterpart to bitwise exclusive-or,*

There is: Just as the bitwise `&` operator corresponds to the the boolean `&&` operator, so does the bitwise `^` operator correspond to the boolean `!=` operator.

It would be redundant to introduce another name for `!=`, when `!=` already exists for booleans.

Jan 2, 2017, 10:10 PM (/post/159129) reply quote

Reply (https://leetcode.com/problems/sum-of-two-integers)

(?page=1) (?page=1) 1 / 2 (?page=2) (?page=2)