### **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 condition;

}

##### **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!

And 运算

and运算通常用于二进制的取[位操作](https://baike.baidu.com/item/%E4%BD%8D%E6%93%8D%E4%BD%9C)，例如一个数 and 1的结果就是取[二进制](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6)的最末位。这可以用来判断一个整数的奇偶，二进制的最末位为0表示该数为[偶数](https://baike.baidu.com/item/%E5%81%B6%E6%95%B0)，最末位为1表示该数为奇数

or 运算

or运算通常用于二进制特定位上的无条件[赋值](https://baike.baidu.com/item/%E8%B5%8B%E5%80%BC)，例如一个数or 1的结果就是把二进制最末位强行变成1。如果需要把二进制最末位变成0，对这个数or 1之后再减一就可以了，其实际意义就是把这个数强行变成最[接近](https://baike.baidu.com/item/%E6%8E%A5%E8%BF%91/1356208)的偶数

 xor运算

xor运算的逆运算是它本身，也就是说两次异或同一个数最后结果不变，即（a xor b) xor b = a。xor运算可以用于简单的加密

加法和减法互为逆运算，并且加法满足交换律。把#换成+，把@换成-，我们可以写出一个不需要临时变量的swap过程（Pascal）。

procedure swap(var a,b:longint);

begin

a:=a + b;

b:=a - b;

a:=a - b;

end;

procedure swap(var a,b:longint);

begin

a:=a xor b;

b:=a xor b;

a:=a xor b;

end;

 not运算 ~ ===

not运算的定义是把内存中的0和1全部取反。使用not运算时要格外小心，你需要注意整数类型有没有符号。如果not的对象是[无符号整数](https://baike.baidu.com/item/%E6%97%A0%E7%AC%A6%E5%8F%B7%E6%95%B4%E6%95%B0)（不能表示负数），那么得到的值就是它与该类型[上界](https://baike.baidu.com/item/%E4%B8%8A%E7%95%8C)的差，因为无符号类型的数是用00到$FFFF依次表示的。下面的两个程序（仅语言不同）均返回65435。

shl运算 << ===

a shl b就表示把a转为二进制后左移b位（在后面添b个0）。例如100的二进制为1100100，而110010000转成十进制是400，那么100 shl 2 = 400。可以看出，a shl b的值实际上就是a乘以2的b次方，因为在二进制数后添一个0就相当于该数乘以2。

通常认为a shl 1比a \* 2更快，因为前者是更底层一些的操作。因此程序中乘以2的操作请尽量用左移一位来代替

shr运算 >> ===

和shl相似，a shr b表示二进制右移b位（去掉末b位），相当于a除以2的b次方（取整）。我们也经常用shr 1来代替div 2，比如[二分查找](https://baike.baidu.com/item/%E4%BA%8C%E5%88%86%E6%9F%A5%E6%89%BE)、堆的插入操作等等

优先级

[编辑](javascript:;)

C语言中位运算符之间，按优先级顺序排列为

|  |  |
| --- | --- |
| 1 | ~ |
| 2 | <<、>> |
| 3 | & |
| 4 | ^ |
| 5 | | |
| 6 | &=、^=、|=、<<=、>>= |

The following two code samples, written in the programming language [C++](https://en.wikipedia.org/wiki/C%2B%2B), both determine if the given unsigned integer **x** is a [power of two](https://en.wikipedia.org/wiki/Power_of_two).

*// The obvious method*

unsigned int x = ...;

bool isPowerOfTwo;

**if** (x > 0) {

*/\* Divide by two as long as the next division is an integer,*

*and if it isn't, check if the number is 1 (meaning the number is*

*some power of two) \*/*

**while** ((x % 2) == 0) {

x = x / 2;

}

isPowerOfTwo = (x==1);

}

**else** { *// zero is never a power of two*

isPowerOfTwo = false;

}

*// A method using bit manipulation*

bool isPowerOfTwo = x && !(x & (x - 1));

The second method uses the fact that powers of two have one and only one bit set in their binary representation:

x == 0...010...0

x-1 == 0...001...1

x & (x-1) == 0...000...0

If the number is neither zero nor a power of two, it will have '1' in more than one place:

x == 0...1...010...0

x-1 == 0...1...001...1

x & (x-1) == 0...1...000...0