Bit Manipulation

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 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;
}
```