BITWISE

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;
}
```

| 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;
}
```

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;
}
```