This repository Search Pull requests Issues Gist

huyilong / **OOP-Design**

Unwatch 1 Unstar 1 Fork 0

Code Issues 0 Pull requests 0 Wiki Pulse Graphs Settings

Branch: master OOP-Design / ***BitManipulateSummary.java Find file Copy path

huyilong SUMMARY - ALLLL **Bit** Problems de61499 2 minutes ago

1 contributor

253 lines (217 sloc) 8.94 KB Raw Blame History

```
Divide two integers without using multiplication, division and mod operator.
最直观的方法是，用被除数逐个的减去除数，直到被除数小于0。这样做会超时。
那么如果每次不仅仅减去1个除数，计算速度就会增加，但是题目不能使用乘法，因此不能减去k*除数，
我们可以对除数进行左移位操作，这样每次相当于减去2^k个除数，
如何确定k呢，只要使 (2^k)*除数 <=  当前被除数 <(2^(k+1))*除数.
因为这个方法的迭代次数是按2的幂直到超过结果，所以时间复杂度为O(logn)。
class Solution {
public:
    int divide(int dividend, int divisor) {
        unsigned int divd = dividend, divs = divisor;//使用unsigned防止-2147483648符号取反后溢出
        if(divisor < 0)divs = -divs;//负数全部转化为正数
        if(dividend < 0)divd = -divd;

        int res = 0;
        while(divd >= divs)
        {
            long long a = divs;//使用long long防止移位溢出
            int i;
            for(i = 1; a <= divd; i++)
                a <<= 1;
            res += (1 << (i-2));
            divd -= (divs << (i-2));
        }
        boolean isNeg = (dividend^divisor)>>>31 == 1;
        //sign is different?
        return isNeg ? -res : res;
    }
};

那么有没有办法优化化呢？ 这个我们就得使用位运算。我们知道任何一个整数可以表示成以2的幂为底的一组基
的线性组合，即num=a_0*2^0+a_1*2^1+a_2*2^2+...+a_n*2^n。基于以上这个公式以及左移一位相当
于乘以2，我们先让除数左移直到大于被除数之前得到一个最大的基。然后接下来我们每次尝试减去这个基，
如果可以则结果增加加2^k,然后基继续右移迭代，直到基为0为止。因为这个方法的迭代次数是按2的幂直到
超过结果，所以时间复杂度为O(logn)。代码如下:
public int divide(int dividend, int divisor) {
    if(divisor == 0)
    {
        return Integer.MAX_VALUE;
    }
    boolean isNeg = (dividend^divisor)>>>31 == 1;
    int res = 0;
    if(dividend == Integer.MIN_VALUE)
    {
        dividend += Math.abs(divisor);
        if(divisor == -1)
        {
            return Integer.MAX_VALUE;
        }
        res++;
    }
    if(divisor == Integer.MIN_VALUE)
    {
        return res;
    }
    dividend = Math.abs(dividend);
    divisor = Math.abs(divisor);
    int digit = 0;
    while(divisor <= (dividend>>1))
    {
        divisor <<= 1;
        digit++;
    }
    while(digit>=0)
    {
        if(dividend>=divisor)
        {
            res += 1<<digit;
            dividend -= divisor;
        }
        divisor >>= 1;
        digit--;
    }
    return isNeg?-res:res;
}

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code,
rint the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return [0,1,3,2]. Its gray code sequence is:
00 - 0
01 - 1
11 - 3
10 - 2
Note:
For a given n, a gray code sequence is not uniquely defined.
For example, [0,2,3,1] is also a valid gray code sequence according to the above definition.
 // Binary to grey code
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int> res;
        for (int i = 0; i < pow(2,n); ++i) {
            res.push_back((i >> 1) ^ i);
        }
        return res;
    }
};

可以看到n位的格雷码由两部分构成，一部分是n-1位格雷码，再加上 1<<(n-1)和n-1位格雷码的逆序
（整个格雷码逆序0132变成2310这种）的和。

1位格雷码有两个码字
(n+1)位格雷码中的前2^n个码字等于n位格雷码的码字，按顺序书写，加前缀0
(n+1)位格雷码中的后2^n个码字等于n位格雷码的码字，按逆序书写，加前缀1。

由于是二进制，在最高位加0跟原来的数本质没有改变，所以取得上一位算出的格雷码结果，再加上逆序添1的方法就是当前这位格雷码的结果了。

n = 0时，[0]

n = 1时，[0,1]

n = 2时，[00,01,11,10]

n = 3时，[000,001,011,010,110,111,101,100]

当n=1时，0，1

当n=2时，原来的list 0，1不变，只是前面形式上加了个0变成00，01。然后加数是1<<1为10，依次：10+1=11 10+0=10。结果为：00 01 11 10

当n=3时，原来的list 00,01,11，10（倒序为：10，11，01，00）。加数1<<2为100。倒序相加为：100+10=110，100+11=111,100+01=101，100+00= 100。

最终结果为000 001 011 010 110 111 101 100

public ArrayList<Integer> grayCode(int n) {
        if(n==0) {
            ArrayList<Integer> result = new ArrayList<Integer>();
            result.add(0);
            return result;
        }

        ArrayList<Integer> result = grayCode(n-1);
        int addNumber = 1 << (n-1);
        int originalsize=result.size();

        for(int i=originalsize-1;i>=0;i--) {
            result.add(addNumber + result.get(i));
        }
        return result;
    }
}
```

```

Number of 1 Bits 位1的个数
check whether a bit is 1 we need to use (n & 1) to plus
class Solution {
public:
    int hammingWeight(uint32_t n) {
        int res = 0;
        for (int i = 0; i < 32; ++i) {
            res += (n & 1);
            n = n >> 1;
        }
        return res;
    }
};

对于这道题，我们只需要把要翻转的数从右向左一位位的取出来，然后加到新生成的数的最低位即可，代码如下：
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t res = 0;

        for (int i = 0; i < 32; ++i) {
            if (n & 1 == 1) {
                res = (res << 1) + 1;
            } else {
                res = res << 1;
            }
            n = n >> 1;
        }

        return res;
    }
};


Given an 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?
本来是一道非常简单的题，但是由于加上了时间复杂度必须是O(n)，并且空间复杂度为O(1)，
使得不能用排序方法，也不能使用map数据结构。那么只能另辟蹊径，需要用位操作Bit Operation来解此题，
这个解法如果让我想，肯定想不出来，因为谁会想到用逻辑异或来解题呢。逻辑异或的真值表为：
由于数字在计算机是以二进制存储的，每位上都是0或1，如果我们把两个相同的数字异或，0与0异或是0,
1与1异或也是0，那么我们会得到0。根据这个特点，我们把数组中所有的数字都异或起来，
则每对相同的数字都会得0，然后最后剩下来的数字就是那个只有1次的数字。

class Solution {
public:
    int singleNumber(int A[], int n) {
        int res = A[0];
        for (int i = 1; i < n; ++i) {
                res ^= A[i];
        }
        return res;
    }
};

Given an array of integers, every element appears three times except for one. Find that single one.

Note:
Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?
用3个整数来表示INT的各位的出现次数情况，one表示出现了1次，two表示出现了2次。当出现3次的时候该位清零。最后答案就是one的值。

ones   代表第ith 位只出现一次的掩码变量
twos  代表第ith 位只出现两次次的掩码变量
threes  代表第ith 位只出现三次的掩码变量
class Solution {
public:
    int singleNumber(int A[], int n) {
        int one = 0, two = 0, three = 0;
        for (int i = 0; i < n; ++i) {
            two |= one & A[i];
            one ^= A[i];
            three = one & two;
            one &= ~three;
            two &= ~three;
        }
        return one;
    }
};

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.

Note:
Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

这题还有一种解法，使用位操作Bit Manipulation来解的，用到了异或操作的特性，
相似的题目有Single Number 单独的数字, Single Number II 单独的数字之二和Single Number III
单独的数字之三。那么思路是既然0到n之间少了一个数，我们将这个少了一个数的数组 & 0到n之间完整的数组
异或一下，那么相同的数字都变为0了，剩下的就是少了的那个数字了，参加代码如下：

take this missing array ^ the complete array
the remaining result is the missing one because all equal ones are 0

class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int res = 0;
        //because res is already 0
        //even if it is missing 0 we could return 0
        //so we just directly (i+1) ^ nums[i] make the complete array start i+1
        for (int i = 0; i < nums.size(); ++i) {
            res = res ^ (i + 1) ^ nums[i];
        }
        return res;
    }
};
```