位运算篇

位运算基础知识：

http://www.cnblogs.com/zhangziqiu/archive/2011/03/30/2000333.html

LeetCode 中关于位运算的题目有以下几道：

1. Single Number
2. Single Number II
3. Divide Two Integers
4. Pow(x, n)
5. Subsets

1. Single Number，题目本身要求是找出唯一一个在数组中出现一次的整数，而其他都会出现两次。这里利用到了位运算中异或的性质，就是两个相同的数进行异或会得到 0，并且任何一个数与 0 的异或还是原数。利用上面的性质，只要把数组中的元素一一异或起来，因为出现两次的会互相抵消，最后会只剩下那个出现一次的整数。这个方法只需要一次扫描，即 O(n)的时间复杂度，而空间上也不需要任何额外变量，即 O(1)的空间复杂度。

代码如下：

```
public int singleNumber(int[] A) {
    if(A == null || A.length == 0){
        return 0;
    }
    int result = A[0];
    for(int i = 1; i < A.length; i++){
        result ^= A[i];
    }
    return result;
}
```

2. Single Number II，上面的方法就没办法了，因为出现三次就不能利用异或的性质了。 算法是对每个位出现1的次数进行统计，因为其他元素都会出现三次，所以最终这些位上的1的个数会是3的倍数。如果我们把统计结果的每一位进行取余3，剩下的结果就会剩下那个出现一次的元素。这个方法对于出现k次都是通用的，包括上面的Single Number也可以用

这种方法，不过没有纯位运算的方法效率高。总体只需要对数组进行一次线性扫描，统计完之后每一位进行取余3并且将位数字赋给结果整数，这是一个常量操作（因为整数的位数是固定32位），所以时间复杂度是O(n)。而空间复杂度需要一个32个元素的数组，也是固定的，因而空间复杂度是O(1)。

代码如下：

```
public int singleNumber(int[] A) {
    int[] digits = new int[32];
    for(int i = 0; i < A.length; i++){
        for(int j = 0; j < 32; j++){
            digits[j] += (A[i] >> j) & 1;
        }
    }
    int result = 0;
    for(int i = 0; i < 32; i++){
        result += (digits[i] % 3) << i;
    }
    return result;
}
```

做题时的感悟：

1. digits[i] += (A[j] >> i) & 1(正确) != digits[i] += (1 << i) & A[j](错误)：

我们希望统计i这一位上1的数量，所以A[j]在i位有1则digits[i]加1，否则不变。digits[i] += (A[j] >> i) & 1(正确)的结果只可能为0或者1，符合要求；而digits[i] += (1 << i) & A[j](错误)在i位上有1的时候，结果不为1而是1 << i. e.g.(1 << 3) & 24的结果是8。所以如果我们想用第二种方法，我们可以通过判断结果是否为0来决定是否为digits[i]加1。e.g. if(((1 << i) & A[j]) != 0){ digits[i]++;}。

2. 这道题还有一种解法，思路巧妙，代码简练，细节如下：

What we need to do is to store the number of '1's of every bit. We know a number appears 3 times at most, so we need 2 bits to store that. Now we have 4 state, 00, 01, 10 and 11, but we only need 3 of them.

In this solution, 00, 01 and 10 are chosen. Let 'ones' represents the first bit, 'twos' represents the second bit. Then we need to set rules for 'ones' and 'twos' so that they act as we hopes. The complete loop is 00->10->01->00(0->1->2->3/0).

• For 'ones', we can get 'ones = ones ^ A[i]; if (twos == 1) then ones = 0', that can be tansformed to 'ones = (ones ^ A[i]) & ~twos'.

• Similarly, for 'twos', we can get 'twos = twos ^ A[i]; if (ones* == 1) then twos = 0' and 'twos = (twos ^ A[i]) & ~ones'. Notice that 'ones*' is the value of 'ones' after calculation, that is why twos is calculated later.

代码如下：

```
public int singleNumber(int[] A) {
    int ones = 0;
    int twos = 0;
    for(int i = 0; i < A.length; i++){
        ones = (A[i] ^ ones) & (~twos);
        twos = (A[i] ^ twos) & (~ones);
    }
    return ones;
}
```

3. Divide Two Integers，对于整数处理问题，比较重要的注意点在于符号和处理越界的问题。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;
    }
    int res = 0;
    if(dividend == Integer.MIN_VALUE){
        if(divisor == -1){
            return Integer.MAX_VALUE;
        }
        res = 1;
        dividend += Math.abs(divisor);
    }
```

```
        if(divisor == Integer.MIN_VALUE){
            return res;
        }
        boolean isNeg = ((dividend ^ divisor) >>> 31) == 1;
        dividend = Math.abs(dividend);
        divisor = Math.abs(divisor);
        int digit = 0;
        while(divisor <= (dividend >> 1)){
            divisor <<= 1;
            digit++;
        }
        while(digit >= 0){
            if(dividend >= divisor){
                dividend -= divisor;
                res += (1 << digit);
            }
            divisor >>= 1;
            digit--;
        }
        return isNeg ? - res : res;
    }
```

做题时的感悟:

1.判断两个数乘积为正或者负可以使用boolean isNeg = ((dividend^divisor)>>>31==1);这种方法还可以避免越界问题。

2. 对于int类型最小的整数比最大的整数绝对值大1，所以如果要取绝对值进行统一处理，那么就要单独处理一下最小整数的情况，上面代码的做法是把它加一个除数让他可以取绝对值。

4. Pow(x, n)，一般来说数值计算的题目可以用两种方法来解，一种是以2为基进行位处理的方法，另一种是用二分法。这道题这两种方法都可以解。

第一种方法在Divide Two Integers使用过，就是把n看成是以2为基的位构成的，因此每一位是对应x的一个幂数，然后迭代直到n到最高位。比如说第一位对应x，第二位对应x*x,第三位对应x^4,...,第k位对应x^(2^(k-1)),可以看出后面一位对应的数等于前面一位对应数的平方，所以可以进行迭代。因为迭代次数等于n的位数，所以算法的时间复杂度是O(logn)。

代码如下：

```
public double pow(double x, int n) {
    if(n == 0){
        return 1.0;
    }
    double res = 1.0;
    if(n < 0){
        if(x >= 1.0 / Double.MAX_VALUE || x <= 1.0 / Double.MIN_VALUE){
            x = 1.0 / x;
        } else {
            return Double.MAX_VALUE;
        }
        if(n == Integer.MIN_VALUE){
            res *= x;
            n++;
        }
    }
    n = Math.abs(n);
    boolean isNeg = false;
    if(n % 2 == 1 && x < 0){
        isNeg = true;
    }
    x = Math.abs(x);
    while(n > 0){
        if((n & 1) == 1){
            if(res > Double.MAX_VALUE / x){
                return Double.MAX_VALUE;
            }
            res *= x;
        }
        x *= x;
        n >>= 1;
    }
    return isNeg ? -res : res;
}
```

以上代码中处理了很多边界情况，这也是数值计算题目比较麻烦的地方。比如一开始为了能够求倒数，我们得判断倒数是否越界，后面在求指数的过程中我们也得检查有没有越界，其实，只要能使结果值变大的运算，都应该考虑越界的问题。所以一般来说求的时候都先

转换为正数，这样可以避免需要双向判断（就是根据符号做两种判断）。接下来我们介绍二分法的解法，如同我们在 Sqrt(x)的方法。不过这道题用递归来解比较容易理解，把 x 的 n 次方划分成两个 x 的 n/2 次方相乘，然后递归求解子问题，结束条件是 n 为 0 返回 1。因为是对 n 进行二分，算法复杂度和上面方法一样，也是 O(logn)。

代码如下：

```
public double pow(double x, int n) {
    if(n == 0){
        return 1.0;
    }
    double half = pow(x, n / 2);
    if(n % 2 == 0){
        return half * half;
    }
    if(n > 0){
        return half * half * x;
    } else {
        return half * half / x;
    }
}
```

以上代码比较简洁，不过这里有个问题是没有做越界的判断，因为这里没有统一符号，所以越界判断分的情况比较多，不过具体也就是在做乘除法之前判断这些值会不会越界，有兴趣的朋友可以自己填充上，这里就不写太啰嗦的代码了。不过实际应用中健壮性还是比较重要的，而且递归毕竟会占用递归栈的空间，所以更推荐第一种解法。

做题时的感悟：

1. 如果指数为负数，需要先求x的倒数1 / x，需要判断越界，

x >= 1.0 / Double.MAX_VALUE || x <= 1.0 / -Double.MAX_VALUE

而且在判断的时候要使用除法，这样可以避免乘法越界

2. 整数取绝对值前需要单独处理Integer.MIN_VALUE，因为它比Integer.MAX_VALUE大一，所以不处理会越界。

3. 乘的时候也需要判断是否越界，res > Double.MAX_VALUE / x。

5. Subsets，求子集问题是经典的NP问题，复杂度上我们就无法强求了，肯定是非多项式量级的。一般来说这个问题有两种解法：递归和非递归。

我们先来说说递归解法，主要递推关系就是假设函数返回递归集合，现在加入一个新的数字，我们如何得到包含新数字的所有子集。其实就是在原有的集合中对每集合中的每个元素都加入新元素得到子集，然后放入原有集合中（原来的集合中的元素不用删除，因为他们也是合法子集）。而结束条件就是如果没有元素就返回空集（注意空集不是null，而是没有元素的数组）就可以了。时间和空间都是取决于结果的数量，也就是O(2^n)。

代码如下：

```
// Solution 1 - Recursion
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
    if(S == null || S.length == 0){
        return null;
    }
    Arrays.sort(S);
    return helper(S, S.length - 1);
}

private ArrayList<ArrayList<Integer>> helper(int[] S, int idx){
    if(idx == -1){
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> item = new ArrayList<Integer>();
        res.add(item);
        return res;
    }
    ArrayList<ArrayList<Integer>> res = helper(S, idx - 1);
    int size = res.size();
    for(int i = 0; i < size; i++){
        if(idx > 0 && S[idx] == S[idx - 1]){
            continue;
        }
        ArrayList<Integer> item = new ArrayList<Integer>(res.get(i));
        item.add(S[idx]);
        res.add(item);
    }
    return res;
}
```

其实非递归解法的思路和递归是一样的，只是没有回溯过程，也就是自无到有的一个个元素加进来，然后构造新的子集加入结果集中。

代码如下：

```
// Solution 2 - Iteration
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(S == null || S.length == 0){
        return res;
    }
    Arrays.sort(S);
    ArrayList<Integer> item = new ArrayList<Integer>();
    res.add(item);
    for(int i = 0; i < S.length; i++){
        int size = res.size();
        for(int j = 0; j < size; j++){
            item = new ArrayList<Integer>(res.get(j));
            item.add(S[i]);
            res.add(item);
        }
    }
    return res;
}
```

这道题因为没有重复的元素，所以还有一种特别的做法，就是用位运算 - Bit Manipulation。思路非常巧妙，值得借鉴。

代码如下：

```
// Solution 3 - Bit Manipulation
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
    if(S == null || S.length == 0){
        return null;
    }
    Arrays.sort(S);
    int total = 1 << S.length;
    ArrayList<ArrayList<Integer>> res = new
     ArrayList<ArrayList<Integer>>(total);
    for(int i = 0; i < total; i++){
        ArrayList<Integer> item = new ArrayList<Integer>();
        for(int j = 0; j < S.length; j++){
```

```
                if(((i >> j) & 1) == 1){
                    item.add(S[j]);
                }
            }
            res.add(item);
        }
        return res;
    }
```

做题时的感悟:

递归:

1. 正如大神所说，没有元素的时候要返回空集，所以在创建res数组的时候，要添加一个空的item数组进去，方便以后的递归计算。

2. 这道题一定要预先获得res的size，int size = res.size()；然后在循环中用size来限制i，否则我们不停向res中添加数组，res的size不停增大，永远到达不了res.size()会变成死循环。

迭代:

需要注意的地方与递归相同，没有特别的地方。

Bit Manipulation:

1. 因为1个元素只有在和不在结果中2种可能，所以结果数量total为2^n，即1 << S.length。res也可以根据total来创建，使结果res更省空间且效率更高。

2. i中的S.length位，每一位对应S中1个元素是否出现。所以可以用((i >> j) & 1) == 1来判断这个数是否出现在结果集中。非常巧妙。