有关基础类型的集合运算和数值运算

基础数据类型稍加延伸，就可以得到新的算法题变种，集合运算和整形数值运算可以跟整形的二进制表现形式进行紧密关联。我们看看两道相关的面试算法题

给定一个集合 S，要求打印出集合 S 的所有子集，子集中的元素用逗号隔开，假设集合 S 的内容为:
S = {“A”, ”B” , “C”}, 那么 S 的所有子集为：

A, B , C
A, B
A, C
B, C
A
B
C
Null
S 本身和空集都可以认为是 S 的子集。

这是一道相当常见的面试算法题，在我的面试经验中，这道题遇到过若干次，面试时被考到的几率是很大的。

集合以及它的子集如何与二进制联系起来呢，二进制不就是一系列 0 和 1 的组合吗？大家看到 0 和 1 是否能有所启发。对应 S 的一个子集，S 中的某个元素要不存在于该子集中，要不不存在与该子集中。如果存在于子集中，我们用 1 表示，不存在用 0 表示，这样看来，子集元素似乎跟二进制有很大的关联。我们以题目中的例子为例，S 含有三个元素，那么我们让 S 对应于拥有三个位长的二进制数：

“A”	“B”	“C”
b	b	b

b 代表一个二进制数，可以取值 0 或 1，如果元素”A”在给定子集中，那么它对应的 b 取值 1，如果不在子集中，b 取值 0，这样一来，子集{“A”, “B”} 对应的二进制数表示为 110. 如此我们就得到了一个子集与一个二进制数的对应关系，那么反过来，给定一个二进制数，我们就得到一个相应的子集，从而题目中的子集可以有如下对应关系:

1 1 1(7) -> A B C

1 1 0 (6)-> A B

1 0 1 (5)-> A C

0 1 1 (3)-> B C

1 0 0 (4)-> A

0 1 0 (2)-> B

0 0 1 (1)-> C

0 0 0 (0)-> null

括号里的数是二进制所对应的十进制数，这样我们就得到了一个有效算法，假定集合 S 有 n 个元素，那么我们就构造 n 个位长的二进制数，遍历 n 个位长的二进制数的所有可能情况，根据二进制数上每个位上是 0 还是 1 来决定对应元素是否在该子集中。遍历所有可能的二进制数也不是难事，以例子为例，当集合元素为 3 时，集合对应的二进制数长度是 3，因此我们要遍历 3 个位长的二进制数所对应的所有情况，3 个位长对应的最大整数是 7(111)，那么 0 到 7 间的所有整数，转换为对应的二进制数就可以得到位长为 3 的二进制数的所有可能情况，

最终算法如下：

```
public void handleAllSubSet(String[] set) {
    int len = set.length;
    int val = 0;
    for (int i = 0; i < len; i++) {
```

```
            val |= (1<<i);
        }

        while (val >= 0) {
            printSetByBinary(val, set);
            val--;
        }
    }
```

在 handleAllSubSet 中，开始的 for 循环目的是根据集合中元素的个数构造相应位长的二进制数，然后根据该二进制数所有可能的取值情况来打印相应的子集元素。printSetByBinary 的实现留个大家作业，它的具体实现，我在稍后的代码调试演示中会详细讲解。

一个 n 位长的二进制数，它所有可能的取值情况有 2^n 种，因此我们算法的复杂度是(2^n)，这种算法效率是相当低下的，这个问题，其目的是考察思路，因此算法效率不是重点。

接下来，我们再看看有关基础数据类型在数值运算方面的算法题。

两个正整数 x, y 的最大公约数,是最大的整数 d, d 必须满足 d | x 并且 d | y, 其中 a | b 表示 b 是 a 的倍数,也就是 b mod a = 0.
要求设计一个算法来计算最大公约数，算法中不能使用乘法，除法和求余运算。

这道题有一定的难度，求最大公约数的算法，最有名的是欧几里得算法，它的做法如下:
假设 a > b, d = a %b, 那么 a,b 的最大公约数等于 b, d 的最大公约数，于是求 a b 的最大公约数可以转换为求 b d 的最大公约数，欧几里得算法的实现如下：

```
int  gcd( a, b) {
    if (a % b == 0) {
        return b;
    }
    d = a % b;
    return gcd(b , d);
}
```

欧几里得算法的正确性和效率在此不做证明，大家可以到网上搜到很多相关答案。

现在题目要求的是，不能使用乘法，除法和求余运算，因此上面算法中 a % b 这样的运算就不符合要求。如果我们有办法把 a % b 的结果通过其他符合要求的运算来获取的话，那么套入上面的欧几里得算法就可以了。

符合题目要求的运算有哪些呢，有加法，减法和位移操作，我们看看如何使用上面的运算实现求余。

由于 a > b, 我们有 a = k * b + d. d 就是我们想要得到的结果。我们用一个具体实例来看看，a = 23, b = 4,有：

23 = 5 * 4 + 3

那么 k = 5. 我们看看 5 的二进制表示是 0b101,于是：

5 = 2^2 + 2^0

红色的 2 是红色 1 的下标，绿色的 0 是绿色的 1 的下标。
我们用一个数组 T[]来存储 k 的二进制形式中 1 的下标，那么对应于 5， 数组 T 的内容就是 T = {2, 0}.

于是上面的等式又可以转换一下：

5 = 2 ^ 2 + 2 ^ 0

= 2 ^ T[1] + 2^T[0]

于是：

23 = 5 * 4 + 3

=(2^T[1] + 2^T[0]) * 4 + 3

= 4 * 2^T[1] + 4*2^T[0] + 3

我们知道，一个数乘以一个 2 的幂，就相当于将该数左移若干位，于是上面的等式又可以进一步推导为：

23 = 4 << T[1] + 4 << T[0] + 3

3 = 23 - (4 << T[1] + 4 << T[0])

3 就是 23 除以 4 后的余数，也就是说通过上面的转换，我们就可以通过减法和位移运算求得两数的余数。

接下来的问题是如何找到数组 T.假设 T 的长度为 n

不难理解：

2 << (T[n-1] + 1) > k > 2<<T[n-1]

根据给定的例子 T 数组的长度是 2，于是有：

T[1] = 2

2 << (T[1] + 1) > 5 > 2<<T[1].

特别有：（ 2 << (T[n-1] + 1) ）>= k + 1.当 k 的二进制形式中，所有位都是 1 的时候上面的等号才会成立，例如：

k = 7 = 0b111 , T[] = {2, 1, 0}, T[n-1] = T[2] = 2;

k + 1 = 8 = 0b1000 = 2 << 3 = (2 << （T[n-1] + 1））

由于 d 是 a 除 b 的余数， 所以 d >= 0, d < b, 于是有：

a = k * b + d < (k+1) * b <= b<<(T[n-1] + 1).

由于 ：

k = 2 << T[n-1] + … 2 << T[0] >= (2 << T[n-1]).

a >= b*k >= b * 2<<T[n-1] = b << T[n-1]

结合起来就有：

b<<(T[n-1] + 1) >= a >= b << T[n-1]

于是只要一个整数 k 满足 ：

b << k >= a, 但是 b << (k-1) <= a,

那么 k - 1 就等于 T[n-1]

于是，要求得 T[n-1] 我们可以这么做：

```
int k = 0;
while ( (b << k) <= a) {
    k++;
}
k--;
```

上面的代码，while 循环出来后，k 的值就是 T[n-1] + 1, 然后执行 k--后，k 的值就是 T[n-1]. 大家可以把上面的代码带入到给定的例子中，看看得到的 k 是不是等于 T[1].

接下来，有了 T[n-1]，如何计算 T[n-2], … T[0] 呢。

根据等式：

a = k*b + d = b<< T[n-1] +… b<<T[0] + d

由于 T[n-1]我们算出来了，于是可以算出 b<<T[n-1].把上面式子做变换：

a’ = a - b <<T[n-1] = b<<T[n-2] +… b<<T[0] + d

于是有：

b<< (T[n-2] + 1) >= a’ >= b << T[n-2]

由于 T[n-1] >= T[n-2] + 1; (为什么)，所以：

b << T[n-1] >= a’ >= b<<T[n-2]

所以，如果我们令 k = T[n-1]，然后每次让 k 减一，一旦 k 满足：

(b << k) >= a’，a’ >= (b << (k-1))
那么 ， k-1 就等于 T[n-2].

由此， T[n-3]…T[0]，可以以此类推求出来.

于是我们就有了求余算法:

```
int  module(int a, int b) {

    ArrayList<Integer> T = new ArrayList<Integer>();
    int k = 0;
    while ( (b << k) <= a) {
       k++;
    }
    k--;
    T.add(k); //T[n-1]

    //a’  = a  -  b<<T[n-1]
    int a_prime = a- (b << T.get(T.size()  -  1));

    do {
        //T[n-2], … T[0];
        while ((b<<k) > a_prime) {
            k--;
        }
        T.add(k);
    a_prime = a_prime - (b << T.get(T.size() -  1));

    }while (a_prime > b);
```

```
    // d = a  -  (b<<T[n-1] + b<<T[n-2]…)
    for (int i = 0; i < T.size(); i++) {
        a -= (b << T.get(i));
    }

    return a;
}
```

我会通过代码调试的方式给大家展现上面代码的逻辑，如果此时还不理解，那么看过代码调试后，应该会清楚很多。于是将求余下算法套入欧几里得算法就有:

```
int gcd(int a, int b) {
    int d = module(a, b);
    if (d == 0) {
        return b;
    }

    return gcd(a,b);
}
```

如果a, b 的二进制表示形式有n个位长,在module函数中，我们需第一个 while 循环的次数最多不超过 n, 第二个 while 循环，虽然有两个循环间套，但其循环次数也不超过 n（为什么?）. gcd 递归调用的次数最多是 n 次，也就是说 module 最多被调用 n 次，因此算法的总体复杂度是 O(n^2).

习题：

给定两个正整数 x, y, 只能使用加法，减法和乘法来计算 x / y 的值。