```
                                num = top.equals("*") ? (cur * num) : (cur / num);
                        }
                }
                deq.addLast(String.valueOf(num));
        }

        public int getNum(Deque<String> deq) {
                int res = 0;
                boolean add = true;
                String cur = null;
                int num = 0;
                while (!deq.isEmpty()) {
                        cur = deq.pollFirst();
                        if (cur.equals("+")) {
                                add = true;
                        } else if (cur.equals("-")) {
                                add = false;
                        } else {
                                num = Integer.valueOf(cur);
                                res += add ? num : (-num);
                        }
                }
                return res;
        }
```

0 左边必有 1 的二进制字符串数量

【题目】

给定一个整数 N，求由"0"字符与"1"字符组成的长度为 N 的所有字符串中，满足"0"字符的左边必有"1"字符的字符串数量。

【举例】

N=1。只由"0"与"1"组成，长度为 1 的所有字符串："0"、"1"。只有字符串"1"满足要求，所以返回 1。

N=2。只由"0"与"1"组成，长度为 2 的所有字符串为："00"、"01"、"10"、"11"。只有字符串"10"和"11"满足要求，所以返回 2。

N=3。只由"0"与"1"组成，长度为 3 的所有字符串为："000"、"001"、"010"、"011"、"100"、"101"、"110"、"111"。字符串"101"、"110"、"111"满足要求，所以返回 3。

【难度】

校 ★★★☆

【解答】

先说一种最暴力的方法，就是检查每一个长度为 N 的二进制字符串，看有多少符合要求。一个长度为 N 的二进制字符串，检查是否符合要求的时间复杂度为 $O(N)$，长度为 N 的二进制字符串数量为 $O(2^N)$，所以该方法整体的时间复杂度为 $O(2^N \times N)$，本书不再详述。

$O(2^N)$的方法。假设第 0 位的字符为最高位字符，很明显，第 0 位的字符不能为'0'。假设 $p(i)$表示 0~i-1 位置上的字符已经确定，这一段符合要求且第 i-1 位置的字符为'1'时，如果穷举 i~N-1 位置上的所有情况会产生多少种符合要求的字符串。比如 N=5，$p(3)$表示 0~2 位置上的字符已经确定，这一段符合要求且位置 2 上的字符为'1'时，假设为"101.."。在这种情况下，穷举 3~4 位置所有可能的情况会产生多少种符合要求的字符串，因为只有"10101"、"10110"和"10111"，所以 $p(3)$=3。也可以假设前三位是"111..", $p(3)$同样等于 3。有了 $p(i)$的定义，同时知道不管 N 是多少，最高位的字符只能为'1'，那么只要求出 $p(1)$就是所有符合要求的字符串数量。

那到底 $p(i)$应该怎么求呢？根据 $p(i)$的定义，在位置 i-1 的字符已经为'1'的情况下，位置 i 的字符可以是'1'，也可以是'0'。如果位置 i 的字符是'1'，那么穷举剩下字符的所有可能性，并且符合要求的字符串数量就是 $p(i+1)$的值。如果位置 i 的字符是'0'，那么位置 i+1 的字符必须是'1'，穷举剩下字符的所有可能性，符合要求的字符串数量就是 $p(i+2)$的值。所以 $p(i)=p(i+1)+p(i+2)$。$p(N-1)$表示除了最后位置的字符，前面的子串全符合要求，并且倒数第二个字符为'1'，此时剩下的最后一个字符既可以是'1'，也可以是'0'，所以 $p(N-1)=2$。$p(N)$表示所有的字符串已经完全确定，并且符合要求，最后一个字符(N-1)为'1'，所以，此时符合要求的字符串数量就是 0~N-1 的全体，而不再有后续的可能性，所以 $p(N)=1$。即 $p(i)$如下：

$i < N-1$ 时，$p(i) = p(i+1)+p(i+2)$

$i = N-1$ 时，$p(i) = 2$　　斐波那契数列

$i = N$ 时，$p(i) = 1$

很明显，可以写成时间复杂度为 $O(2^N)$的递归方法。具体请参看如下的 getNum1 方法。

```
public int getNum1(int n) {
        if (n < 1) {
                return 0;
        }
        return process(1, n);
```

```
    }

    public int process(int i, int n) {
        if (i == n - 1) {
            return 2;
        }
        if (i == n) {
            return 1;
        }
        return process(i + 1, n) + process(i + 2, n);
    }
```

根据 $O(2^N)$的方法，当 N 分别为 1，2，3，4，5，6，7，8 时，结算的结果为 1，2，3，5，8，13，21，34。可以看出，这就是一个形如斐波那契数列的结果，唯一的区别就是斐波那契数列的初始项为 1，1。而这个数列的初始项为 1，2。所以可很轻易地写出时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$的方法。具体请参看如下代码中的 getNum2 方法。

```
    public int getNum2(int n) {
        if (n < 1) {
            return 0;
        }
        if (n == 1) {
            return 1;
        }
        int pre = 1;
        int cur = 1;
        int tmp = 0;
        for (int i = 2; i < n + 1; i++) {
            tmp = cur;
            cur += pre;
            pre = tmp;
        }
        return cur;
    }
```

打开了斐波那契数列的这个天窗，我们知道求解斐波那契数列的过程，有时间复杂度为 $O(\log N)$方法就是用矩阵乘法的办法求解，具体解释请参考本书"斐波那契数列的 3 种解法"，这里不再详述。代码实现请参看如下代码中的 getNum3 方法。

```
    public int getNum3(int n) {
        if (n < 1) {
            return 0;
        }
        if (n == 1 || n == 2) {
            return n;
        }
        int[][] base = { { 1, 1 }, { 1, 0 } };
        int[][] res = matrixPower(base, n - 2);
```

```
        return 2 * res[0][0] + res[1][0];
}

public int[][] matrixPower(int[][] m, int p) {
        int[][] res = new int[m.length][m[0].length];
        for (int i = 0; i < res.length; i++) {
                res[i][i] = 1;
        }
        int[][] tmp = m;
        for (; p != 0; p >>= 1) {
                if ((p & 1) != 0) {
                        res = muliMatrix(res, tmp);
                }
                tmp = muliMatrix(tmp, tmp);
        }
        return res;
}

public int[][] muliMatrix(int[][] m1, int[][] m2) {
        int[][] res = new int[m1.length][m2[0].length];
        for (int i = 0; i < m2[0].length; i++) {
                for (int j = 0; j < m1.length; j++) {
                        for (int k = 0; k < m2.length; k++) {
                                res[i][j] += m1[i][k] * m2[k][j];
                        }
                }
        }
        return res;
}
```

拼接所有字符串产生字典顺序最小的大写字符串

【题目】

给定一个字符串类型的数组 strs，请找到一种拼接顺序，使得将所有的字符串拼接起来组成的大写字符串是所有可能性中字典顺序最小的，并返回这个大写字符串。

【举例】

strs=["abc"，"de"]，可以拼成"abcde"，也可以拼成"deabc"，但前者的字典顺序更小，所以返回"abcde"。

strs=["b"，"ba"]，可以拼成"bba"，也可以拼成"bab"，但后者的字典顺序更小，所以返回"bab"。