```
                    next = tmp;
                }
            }
            return cur;
        }
```

因为是顺序计算，所以 num2 方法的时间复杂度为 $O(N)$，同时只用了 cur、next 和 tmp 进行滚动更新，所以额外空间复杂度为 $O(1)$。但是本题并不能像斐波那契数列问题那样用矩阵乘法的优化方法将时间复杂度优化到 $O(\log N)$，这是因为斐波那契数列是严格的 $f(i)=f(i-1)+f(i-2)$，但是本题并不严格，str[i]的具体情况决定了 $p(i)$是等于 0 还是等于 $p(i+1)$，还是等于 $p(i+1)+p(i+2)$。有状态转移的表达式不可以用矩阵乘法将时间复杂度优化到 $O(\log N)$。但如果 str 只由字符'1'和字符'2'组成，比如"12121121212122"，那么就可以使用矩阵乘法的方法将时间复杂度优化为 $O(\log N)$。因为 str[i]都可以单独转换成字母，str[i..i+1]也都可以一起转换成字母，此时一定有 $p(i)=p(i+1)+p(i+2)$。总之，可以使用矩阵乘法的前提是递归表达式不会发生转移。

表达式得到期望结果的组成种数

【题目】

给定一个只由 0（假）、1(真)、&(逻辑与)、|（逻辑或）和^(异或)五种字符组成的字符串 express，再给定一个布尔值 desired。返回 express 能有多少种组合方式，可以达到 desired 的结果。

【举例】

express="1^0|0|1"，desired=false。

只有 1^((0|0)|1)和 1^(0|(0|1))的组合可以得到 false，返回 2。

express="1"，desired=false。

无组合则可以得到 false，返回 0。

【难度】

校 ★★★☆

【解答】

应该首先判断 express 是否合乎题目要求，比如"1^"和"10"，都不是有效的表达式。总结起来有以下三个判断标准：

- 表达式的长度必须是奇数。
- 表达式下标为偶数位置的字符一定是'0'或者'1'。
- 表达式下标为奇数位置的字符一定是'&'或'|'或'^'。

只要符合上述三个标准，表达式必然是有效的。具体参看如下代码中的 isValid 方法。

```
public boolean isValid(char[] exp) {
        if ((exp.length & 1) == 0) {
                return false;
        }
        for (int i = 0; i < exp.length; i = i + 2) {
                if ((exp[i] != '1') && (exp[i] != '0')) {
                        return false;
                }
        }
        for (int i = 1; i < exp.length; i = i + 2) {
                if ((exp[i] != '&') && (exp[i] != '|') && (exp[i] != '^')) {
                        return false;
                }
        }
        return true;
}
```

暴力递归方法。在判断 express 符合标准之后，将 express 划分成左右两个部分，求出各种划分的情况下，能得到 desired 的种数是多少。以本题的例子进行举例说明，express 为"1^0|0|1"，desired 为 false，总的种数求法如下：

- 第 1 个划分为'^'，左部分为"1"，右部分为"0|0|1"，因为当前划分的逻辑符号为^，所以要想在此划分下得到 false，包含的可能性有两种：左部分为真，右部分为真；左部分为假，右部分为假。

结果 1 = 左部分为真的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

- 第 2 个划分为'|'，左部分为"1^0"，右部分为"0|1"，因为当前划分的逻辑符号为|，所以要想在此划分下得到 false，包含的可能性只有一种，即左部分为假，右部分为假。

结果 2 = 左部分为假的种数 × 右部分为假的种数。

- 第 3 个划分为'|'，左部分为"1^0|0"，右部分为"1"，因为当前划分的逻辑符号为|，

所以结果 3 = 左部分为假的种数 × 右部分为假的种数。

- 结果 1+结果 2+结果 3 就是总的种数，也就是说，一个字符串中有几个逻辑符号，就有多少种划分，把每种划分能够得到最终 desired 值的种数全加起来，就是总的种数。

现在来系统地总结一下划分符号和 desired 的情况。

① 划分符号为^、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数。

② 划分符号为^、desired 为 false 的情况下：

种数 = 左部分为真的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

③ 划分符号为&、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为真的种数。

④ 划分符号为&、desired 为 false 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数 + 左部分为假的种数 × 右部分为假的种数。

⑤ 划分符号为|、desired 为 true 的情况下：

种数 = 左部分为真的种数 × 右部分为假的种数 + 左部分为假的种数 × 右部分为真的种数 + 左部分为真的种数 × 右部分为真的种数。

⑥ 划分符号为|、desired 为 false 的情况下：

种数 = 左部分为假的种数 × 右部分为假的种数。

根据如上总结，以 express 中的每一个逻辑符号来划分 express，每种划分都求出各自的种数，再把种数累加起来，就是 express 达到 desired 总的种数。每次划分出的左右两部分递归求解即可。具体过程请参看如下代码中的 num1 方法。

```
public int num1(String express, boolean desired) {
    if (express == null || express.equals("")) {
        return 0;
    }
    char[] exp = express.toCharArray();
    if (!isValid(exp)) {
        return 0;
    }
    return p(exp, desired, 0, exp.length - 1);
}
```

```
public int p(char[] exp, boolean desired, int l, int r) {
    if (l == r) {
        if (exp[l] == '1') {
            return desired ? 1 : 0;
        } else {
            return desired ? 0 : 1;
        }
    }
    int res = 0;
    if (desired) {
        for (int i = l + 1; i < r; i += 2) {
            switch (exp[i]) {
                case '&':
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    break;
                case '|':
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    break;
                case '^':
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    break;
            }
        }
    } else {
        for (int i = l + 1; i < r; i += 2) {
            switch (exp[i]) {
                case '&':
                    res += p(exp, false, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, true, l, i - 1) * p(exp, false, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
                case '|':
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
                case '^':
                    res += p(exp, true, l, i - 1) * p(exp, true, i + 1, r);
                    res += p(exp, false, l, i - 1) * p(exp, false, i + 1, r);
                    break;
            }
        }
    }
    return res;
}
```

一个长度为 N 的 express，假设计算 express[i..j]的过程记为 $p(i,j)$，那么计算 $p(0,N\text{-}1)$需要计算 $p(0,0)$与 $p(1,N\text{-}1)$、$p(0,1)$与 $p(2,N\text{-}1)$...p$(0,i)$与 $p(i\text{+}1,N\text{-}1)$...$p(0,N\text{-}2)$与 $p(N\text{-}1,N\text{-}1)$，起码 $2N$ 种状态。对于每一组 $p(0,i)$与 $p(i\text{+}1,N\text{-}1)$来说，两者相加的划分种数又是 N-1 种，所以

起码要计算 2(N-1)种状态。所以用 num1 方法来计算一个长度为 N 的 express，总的时间复杂度为 $O(N!)$，额外空间复杂度为 $O(N)$，因为函数栈的大小为 N。之所以用暴力递归方法的时间复杂度这么高，是因为每一种状态计算过后没有保存下来，导致重复计算的大量发生。

动态规划的方法。如果 express 长度为 N，生成两个大小为 $N \times N$ 的矩阵 t 和 f，t[j][i]表示 express[j..i]组成 true 的种数，f[j][i]表示 express[j..i]组成 false 的种数。t[j][i]和 f[j][i]的计算方式还是枚举 express[j..i]上的每种划分。具体过程请参看如下代码中的 num2 方法。

```
public int num2(String express, boolean desired) {
    if (express == null || express.equals("")) {
    return 0;
    }
    char[] exp = express.toCharArray();
    if (!isValid(exp)) {
    return 0;
    }
    int[][] t = new int[exp.length][exp.length];
    int[][] f = new int[exp.length][exp.length];
    t[0][0] = exp[0] == '0' ? 0 : 1;
    f[0][0] = exp[0] == '1' ? 0 : 1;
    for (int i = 2; i < exp.length; i += 2) {
    t[i][i] = exp[i] == '0' ? 0 : 1;
    f[i][i] = exp[i] == '1' ? 0 : 1;
    for (int j = i - 2; j >= 0; j -= 2) {
        for (int k = j; k < i; k += 2) {
            if (exp[k + 1] == '&') {
                t[j][i]+=t[j][k] * t[k + 2][i];
                f[j][i]+=(f[j][k] + t[j][k]) * f[k + 2][i] + f[j][k] * t[k + 2][i];
            } else if (exp[k + 1] == '|') {
                t[j][i]+=(f[j][k] + t[j][k]) * t[k + 2][i] + t[j][k] * f[k + 2][i];
                f[j][i]+=f[j][k] * f[k + 2][i];
            } else {
                t[j][i]+=f[j][k] * t[k + 2][i] + t[j][k] * f[k + 2][i];
                f[j][i]+=f[j][k] * f[k + 2][i] + t[j][k] * t[k + 2][i];
            }
        }
    }
    }
    return desired ? t[0][t.length - 1] : f[0][f.length - 1];
}
```

矩阵 t 和 f 的大小为 $N \times N$，每个位置在计算的时候都有枚举的过程，所以动态规划方法的时间复杂度为 $O(N^3)$，额外空间复杂度为 $O(N^2)$。