【第三十四周】一个公式引发的算法学习惨案

1、287. 寻找重复数

1. 二分查找除了对索引二分，还有值域二分
2. 数组元素是 1 - n 中的某一个，出现的位置不确定，但值域是确定的。
3. 对索引二分，一般用于有序数组中找元素，因为索引的大小可以反映值的大小，因此对索引二分即可。
4. 对值域二分。重复数落在 [1, n] ，可以对 [1, n] 这个值域二分查找。
5. mid = (1 + n) / 2，重复数要么落在[1, mid]，要么落在[mid + 1, n]。
6. 遍历原数组，统计 <= mid 的元素个数，记为 k。
7. 如果k > mid，说明有超过 mid 个数落在[1, mid]，但该区间只有 mid 个“坑”，说明重复的数落在[1, mid]。
8. 相反，如果k <= mid，则说明重复数落在[mid + 1, n]。
9. 对重复数所在的区间继续二分，直到区间闭合，重复数就找到了。

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var findDuplicate = function(nums) {
    let left = 1;
    let right = nums.length - 1; //题目注明了：nums.length == n + 1
    while (left < right) {
        const mid = (left + right) >>> 1;  // 求中间索引
        let count = 0;
        for (let i = 0; i < nums.length; i++) {
            if (nums[i] <= mid) {
                count++;
            }
        }
        if (count > mid) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
};
```

2、1175. 质数排列

1. 这个大数不识别，所以需要BigInt方法，但是BigInt方法就非常消耗时间和内存。总之就是先求质数，再求质数与非质数的阶乘，在将两个数相乘
2. 质数是指在大于1的自然数中，除了1和它本身以外不再有其他因数的自然数。
3. 最后要分开乘, 不然 质数排列总数 和 非质数排列总数 一起相乘结果会有偏差

```
/**
 * @param {number} n
 * @return {number}
 */
var numPrimeArrangements = function(n) {
    // 1 判断是否是质数
    let isPrime = num => {
        for(let i=2; i<=Math.sqrt(num); i++) {
            if(num%i === 0) return false;
        }
        return true;
    };
    // 2 获取有几个质数
    let getPrimeNum = num => {
        let primeNum = 0;
        for(let j=2; j<=num; j++) {
            if(isPrime(j)) primeNum++;
        }
        return primeNum;
    }
    // 3 获取阶乘的结果
    let getFactorial = num => {
        let factorial = BigInt(1);
        for(let k=2; k<=num; k++) {
            factorial *= BigInt(k);
        }
        return factorial;
    }
    // 4 取模运算
    let getModNum = num => {
        const modNum = BigInt(10**9 + 7);
        return (num%modNum + modNum) % modNum;
    }
    // 5 获取最终结果
    let primeNum = getPrimeNum(n),
        result = getFactorial(primeNum) * getFactorial(n-primeNum);
    return Number(getModNum(result));
};
```

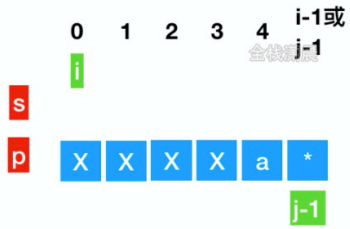
3、1071. 字符串的最大公因子

1. 由题意得知，需要在字符串`str1`和字符串`str2`中找寻一个最大的字符串`x`，使得`x`可以同时除尽字符串`str1`和字符串`str2`。
2. 意味着——字符串`str1`和`str2`内只存在**一或多个**的字符`x`。
3. 字符串`str1`可由`x`个字符串`x`拼接而成，字符串`str2`可由`y`个字符串`x`构成，`x、y`皆大于等于1。
4. 可得公式，str1 + str2 = (x + y)X。那么同理，str2 + str1 = (y + x)X。
5. 所以只要满足str1 + str2 === str2 + str1（Javascript代码）,说明了在字符串str1和str2内必存在字符串X。
6. gcd定理：两个整数的最大公约数等于其中较小的那个数和两数相除余数的最大公约数
   gcd算法：gcd(a,b) = gcd(b,a mod b) (不妨设a>b 且r=a mod b ,r不为0)

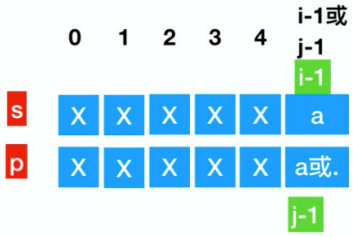
```
/**
 * @param {string} str1
 * @param {string} str2
 * @return {string}
 */
//  最大公约数计算公式
// function gcd(num1, num2) {
//     利用辗转相除法来计算最大公约数
//     return num2 === 0 ? num1 : gcd(num2, num1 % num2);
// }
var gcdOfStrings = function(str1, str2) {
    // 不相等则间接说明了不存在字符串x
    if (str1 + str2 !== str2 + str1) {
        return '';
    }

    // 最大公约数计算公式
    function gcd(num1, num2) {
        // 利用辗转相除法来计算最大公约数，即字符串x在字符串str1中截止的索引位置
        return num2 === 0 ? num1 : gcd(num2, num1 % num2);
    }

    // 截取匹配的字符串
    return str1.substring(0, gcd(str1.length, str2.length));
};
```

4、10. 正则表达式匹配

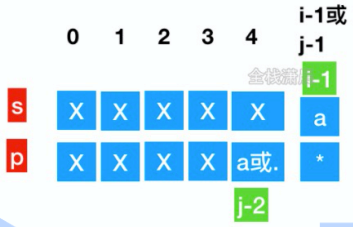
Leetcode:10 //dp[i][j]表示s的前i个字符能否和p的前j个字符匹配



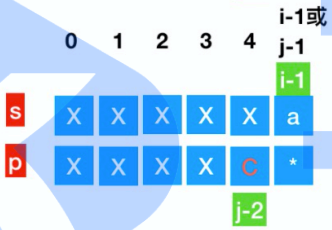
情况1: dp[0][j] = dp[0][j - 2]
i处于0号位置
j处于j-1的位置
当前状态dp[0][j]取决于j向前看2个位置能匹配
相当于*重复0次



情况2: dp[i][j] = dp[i - 1][j - 1];
i处于i-1号位置
j处于j-1的位置
当前位置匹配，即s[i - 1] == p[j - 1] || p[j - 1] == "."
当前状态dp[i][j] 取决于，i和j向前看1个位置能否匹配



情况3: dp[i][j] = dp[i][j - 2] || dp[i][j - 1] || dp[i - 1][j];
i处于i-1号位置
j处于j-2的位置
并且这两个位置相匹配
当前状态dp[i][j]取决于3种case，其中一种能匹配则dp[i][j]就能匹配
case1: dp[i][j - 2]：p向前看2个位置，相当于*重复了0次，
case2: dp[i][j - 1]：p向前看1个位置，相当于*重复了1次
case3: dp[i - 1][j]：s向前看一个位置，相当于*重复了n次



情况4: dp[i][j] = dp[i][j - 2];
i处于i-1号位置
j处于j-2的位置
并且这两个位置不匹配
当前状态dp[i][j]取决于 j向前看2个位置，即*重复0次

```
/**
 * @param {string} s
 * @param {string} p
 * @return {boolean}
 */
//思路：dp[i][j] 表示 s 的前 i 个字符能否和p的前j个字符匹配，分为四种情况，看图
//复杂度：时间复杂度O(mn)，m,n分别是字符串s和p的长度，需要嵌套循环s和p。空间复杂度O(mn)，dp数组所占的空间
//dp[i][j]表示s的前i个字符能否和p的前j个字符匹配
var isMatch = function(s, p) {
    if (s == null || p == null) return false;//极端情况 s和p都是空 返回false

    const sLen = s.length, pLen = p.length;

    const dp = new Array(sLen + 1);//因为位置是从0开始的，第0个位置是空字符串 所以初始化长度是sLen + 1
    for (let i = 0; i < dp.length; i++) {//初始化dp数组
        dp[i] = new Array(pLen + 1).fill(false); // 将项默认为false
    }
    // base case s和p第0个位置是匹配的
```

```
    dp[0][0] = true;
    for (let j = 1; j < pLen + 1; j++) {//初始化dp的第一列，此时s的位置是0
        //情况1:如果p的第j-1个位置是*，则j的状态等于j-2的状态
        //例如: s='' p='a*' 相当于p向前看2个位置如果匹配，则*相当于重复0个字符
        if (p[j - 1] == "*") dp[0][j] = dp[0][j - 2];
    }
    // 迭代
    for (let i = 1; i < sLen + 1; i++) {
        for (let j = 1; j < pLen + 1; j++) {

            //情况2:如果s和p当前字符是相等的 或者p当前位置是. 则当前的dp[i][j] 可由dp[i - 1][j - 1]转移过来
            //当前位置相匹配，则s和p都向前看一位 如果前面所有字符相匹配 则当前位置前面的所有字符也匹配
            //例如: s='XXXa' p='XXX.' 或者 s='XXXa' p='XXXa'
            if (s[i - 1] == p[j - 1] || p[j - 1] == ".") {
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] == "*") {//情况3:进入当前字符不匹配的分支 如果当前p是* 则有可能会匹配
                //s当前位置和p前一个位置相同 或者p前一个位置等于. 则有三种可能
                //其中一种情况能匹配 则当前位置的状态也能匹配
                //dp[i][j - 2]: p向前看2个位置，相当于*重复了0次,
                //dp[i][j - 1]: p向前看1个位置，相当于*重复了1次
                //dp[i - 1][j]: s向前看一个位置，相当于*重复了n次
                //例如 s='XXXa' p='XXXa*'
                if (s[i - 1] == p[j - 2] || p[j - 2] == ".") {
                    dp[i][j] = dp[i][j - 2] || dp[i][j - 1] || dp[i - 1][j];
                } else {
                    //情况4:s当前位置和p前2个位置不匹配，则相当于*重复了0次
                    //例如 s='XXXb' p='XXXa*' 当前位置的状态和p向前看2个位置的状态相同
                    dp[i][j] = dp[i][j - 2];
                }
            }
        }
    }
    return dp[sLen][pLen]; // 长为sLen的s串 是否匹配 长为pLen的p串
};
```

5、49. 字母异位词分组

1. 两个字符串互为字母异位词，当且仅当两个字符串包含的字母相同。同一组字母异位词中的字符串具备相同点，可以使用相同点作为一组字母异位词的标志，使用哈希表存储每一组字母异位词，哈希表的键为一组字母异位词的标志，哈希表的值为一组字母异位词列表。
2. 遍历每个字符串，对于每个字符串，得到该字符串所在的一组字母异位词的标志，将当前字符串加入该组字母异位词的列表中。遍历全部字符串之后，哈希表中的每个键值对即为一组字母异位词。

3. 方法使用【排序】
4. 由于互为字母异位词的两个字符串包含的字母相同，因此对两个字符串分别进行排序之后得到的字符串一定是相同的，故可以将排序之后的字符串作为哈希表的键。

```
/**
 * @param {string[]} strs
 * @return {string[][]}
 */
var groupAnagrams = function(strs) {
    const map = new Map();
    for (let str of strs) {
        let array = Array.from(str);
        array.sort();
        let key = array.toString();
        let list = map.get(key) ? map.get(key) : new Array();
        list.push(str);
        map.set(key, list);
    }
    return Array.from(map.values());
};
```

6、67. 二进制求和

1. 整体思路是将两个字符串较短的用 0 补齐，使得两个字符串长度一致，然后从末尾进行遍历计算，得到最终结果。
2. 本题解中大致思路与上述一致，但由于字符串操作原因，不确定最后的结果是否会多出一位进位，所以会有 2 种处理方式：
3. 第一种，在进行计算时直接拼接字符串，会得到一个反向字符，需要最后再进行翻转
4. 第二种，按照位置给结果字符赋值，最后如果有进位，则在前方进行字符串拼接添加进位

```
/**
 * @param {string} a
 * @param {string} b
 * @return {string}
 */
var addBinary = function(a, b) {
    let ans = "";
    let ca = 0;
    for(let i = a.length - 1, j = b.length - 1;i >= 0 || j >= 0; i--, j--) {
        let sum = ca;
        sum += i >= 0 ? parseInt(a[i]) : 0;
        sum += j >= 0 ? parseInt(b[j]) : 0;
        ans += sum % 2;
        ca = Math.floor(sum / 2);
    }
```

```
    ans += ca == 1 ? ca : "";
    return ans.split('').reverse().join('');
};
```