递归式从 O (N) 简化成 O (logn) 的方法

递归需严格满足递归式，如斐波那契数列`F(n) = F(n-1) + F(n-2)`，必须在递归的每一层中都满足递归式，不存在其他条件分支返回结果`F(n)`

斐波那契数列（leetcode 剑指offer10）

思路1（递归）：

时间复杂度O(n)

```
function fib(n) {
    if (n < 2) {
        return n;
    }

    let a= 0, b = 0, dp = 1;
    for (let i = 2; i <= n; i++) {
        a = b;
        b = dp;
        dp = a + b;
    }

    return dp;
}
```

思路2（简化）：

时间复杂度O(logn)

递归严格满足递归式时，递归式满足如下规律：

递归式：`F(n) = F(n-1) + F(n-2)`

$|F(3), F(2)| = |F(2), F(1)| * \left[\begin{matrix} a & b \ c & d \end{matrix} \right]$

$|F(4), F(3)| = |F(3), F(2)| * \left[\begin{matrix} a & b \ c & d \end{matrix} \right]$

$|F(5), F(4)| = |F(4), F(3)| * \left[\begin{matrix} a & b \ c & d \end{matrix} \right]$

$.....$

$|F(n), F(n-1)| = |F(n-1), F(n-2)| * \left[\begin{matrix} a & b \ c & d \end{matrix} \right]$

因此可以归纳得出：

$|F(n), F(n-1)| = |F(2), F(1)| * \left[\begin{matrix} a & b \ c & d \end{matrix} \right]^{n-2}$

在斐波那契数列中a = 1 b = 1 c = 1 d = 0;

所以$|F(n), F(n-1)| = |1, 1| * \left[\begin{matrix} 1 & 1 \ 1 & 0 \end{matrix} \right]^{n-2}$

只要求解出矩阵matrix的n-2次方就可以得出F(n);

快速求解矩阵n次方（时间复杂度O(logn)）

假设n的二进制为1000101，初始res为单位矩阵，t = matrix;

循环n的二级制位次，每次循环t = t * t，如果n的当前位上为1则res *= t;

如上述列子n = 69，二进制位1000101，每次循环时t分别为m、m^2、m^4、m^8、m^{16}、m^{32}、m^{64}，n在第1、3、7位时位1，则$res=m*m^4*m^{64}=m^{69}$;

```
type matrix = number[][];
function muliMatrix(m1: matrix, m2: matrix): matrix {
    if (m1[0].length !== m2.length) {
        throw new Error("长度不匹配");
    }

    const m = m1.length;
    const p = m2.length;
    const n = m2[0].length;

    const mat = new Array(m).fill(0).map(arr => new Array(n).fill(0));
    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            for (let k = 0; k < p; k++) {
                mat[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }

    return mat;
}
function matrixPower(mat: matrix, n: number): matrix {
    const m = mat.length;
    let res = new Array(m).fill(0).map((arr, index) =>{
        let temp =  new Array(m).fill(0)
        temp[index] = 1;
        return temp;
    });

    while (n > 0) {
        if (n & 1) {
            res = muliMatrix(res, mat);
        }
        mat = muliMatrix(mat, mat);
        n = n  >> 1;
    }

    return res;
}
function fib(n: number): number {
    if (n < 2) {
        return n
```

```
    }

    const mat = [[1,1],[1,0]];
    const matPower = matrixPower(mat, n -2);
    const resultMat = muliMatrix([[1,1]],matPower);
    return resultMat[0][0];
}
```

推广到一般情况

递归式：$F(n) = a*F(n-1)* + *b*F(n-4) ... + c*F(n-m)$

可以转换成（只需要注意n减去最多的数m）：

$|F(n), F(n-1), F(n-2), F(n-3), F(n-4), ... ,F(n-m)| = |F(m), F(m-1),...F(1)|*matrix^{n-m}$，*其中matrix为*mm的矩阵

题目：

农场中有一头母牛，每年会生下一头小母牛，小母牛三年后才能生孩子，问第n年农场有多少头母牛？（假设母牛不会死）

递归式： $F(n) = F(n-1) + F(n-3)$

题目：

农场中有一只母兔子，每年会生两只小母兔，小母兔需要两年才会成熟生孩子，每只兔子五年后就会死掉，问n年之后有多少只兔子?

递归式：$F(n) = F(n-1) + 2*F(n-2) - F(n-5)$

题目一

字符串只由'0'和'1'两种字符构成，

当字符串长度为1时，所有可能的字符串为"0"、"1"；

当字符串长度为2时，所有可能的字符串为"00"、"01"、"10"、"11"；

当字符串长度为3时，所有可能的字符串为"000"、"001"、"010"、"011"、"100"、 "101"、"110"、"111" ...

如果某一个字符串中，只要是出现'0'的位置，左边就靠着'1'，这样的字符串叫作达标字符串。

给定一个正数N，返回所有长度为N的字符串中，达标字符串的数量。 比如，N=3，返回3，因为只有"101"、"110"、"111"达标。

思路1（暴力递归+打表）：

1. 生成所有可能的字符串（2^n）；
2. 遍历每个可能的字符串判断是否达标（n）；
3. 时间复杂度O(2^n*n);
4. 打表可发现符合斐波那契数列的规律；

```
function genStr(n: number): string[] {
    if (n === 1) {
        return ["1", "0"]
    }
    const set: Set<string> = new Set();
    const nextRes = genStr(n-1);
    for (let str of nextRes) {
        set.add('1' + str);
        set.add('0' + str);
        set.add(str + '1');
        set.add(str + '0');
    }
    const res = Array.from(set)
    return res;
}

function standardStrNum(n: number): number {
    if (n < 2) {
        return n;
    }
    let res = 0;
    const allStr = genStr(n);

    for (let str of allStr) {
        let prev = str[0];
        let falg = true;
        for (let s of str) {
            if (s === "0" && prev !== "1") {
                falg = false;
                break
            }
            prev = s;
        }
        res = falg ? res + 1 : res;
    }

    return res;
}
```

思路2（递归）：

1. 如果是达标字符串，一定是以1为开头。所以长度为n的字符串如果为达标字符串一定为`1xxx...`，可能的达标字符串为F(n)（规定F(n)为以1开头、长度为n的达标字符串的个数）；
2. 如果第二个位置为1，字符串为`11xxx...`，此时达标字符串的数量可以看成长度为n-1字符串的达标个数F(n-1);
3. 如果第二个位置为0，字符串为`10xxx...`，此时第三个位置一定为1才能构成达标，所以达标个数为长度n-2字符串的达标个数F(n-2);
4. 递归式为：F(n) = F(n-1) + F(n-2);
5. 简化后时间复杂度O(logn);

题目2

在迷迷糊糊的大草原上，小红捡到了n根木棍，第i根木棍的长度为i，小红现在很开心。想选出其中的三根木棍组成美丽的三角形。

但是小明想捉弄小红，想去掉一些木棍，使得小红任意选三根木棍都不能组成三角形。 请问小明最少去掉多少根木棍呢？ 给定N，返回至少去掉多少根？

思路：

1. 只保留小于n的斐波那契数列中的数，其他的数去掉，这样任意两个数a+b<=c（a < c , b < c）。假设c为斐波那契数列第m项，要想使a+b最大a和b需要为斐波那契数列的第m-1和m-2项，所以a+b=m，故任意小于c的两个数相加不会大于c，无法构成三角形。

题目3（Java的题目）

给定一个字符串，如果该字符串符合人们日常书写一个整数的形式，返回int类型的这个数；如果不符合或者越界返回-1或者报错。

思路：

1. 数字之外只能出现符号"-";
2. 如果有"-"，只能出现在开头且后面必须有数字且不能为0；
3. 如果开头为0，后续必需无数字；
4. 判断数是否越界
    1. 判断的时候用负数接这个值，因为负数的范围比正数大

题目4

牛牛准备参加学校组织的春游, 出发前牛牛准备往背包里装入一些零食,牛牛的背包容量为w。

牛牛家里一共有n袋零食, 第i袋零食体积为v[i]。

牛牛想知道在总体积不超过背包容量的情况下，他一共有多少种零食放法(总体积为0也算一种放法)。

思路（动态规划）：

1. 假设背包容量为N，将问题转换为背包容量为1、2、3、...、N的时候背包刚好装下背包容量的零食所有的方法数总和；
2. 当背包容量为m时，装零食的方法数（动态规划-找零钱问题）：
    1. 零食数组arr，dp[i][j]表示在arr中用0~i的零食组成重量j的方法数，每个零食可以选择要和不要；
    2. dp[i][j] = dp[i-1][j]（第i个零食不要） + dp[i-1][j-arr[i]]（第i个零食要）

```
function compriseWays(arr: number[], w: number): number {
    if (arr == null || arr.length == 0 || w < 0) {
        return 0;
    }
    // dp[i][j]表示用0~i的食物刚好装满j的背包的方法数
    const dp:number[][] = new Array(arr.length).fill(0).map(arr => new
Array(w+1).fill(0));;
    for (let i = 0; i < arr.length; i++) {
        dp[i][0]  = 1;
    }
```

```
    for (let j = 0; j <= w; j++) {
        dp[0][j] = arr[0] <= j ? 2 : 1;
    }

    for (let i = 1; i < arr.length; i++) {
        for (let j = 1; j <= w; j++) {
            dp[i][j] = dp[i-1][j] + (j-arr[i] >= 0 ? dp[i-1][j-arr[i]] : 0);
        }
    }
    return dp[arr.length-1][w];
}
```

题目5

为了找到自己满意的工作，牛牛收集了每种工作的难度和报酬。牛牛选工作的标准是在难度不超过自身能力值的情况下，牛牛选择报酬最高的工作。在牛牛选定了自己的工作后，牛牛的小伙伴们来找牛牛帮忙选工作，牛牛依然使用自己的标准来帮助小伙伴们。牛牛的小伙伴太多了，于是他只好把这个任务交给了你。

```
class Job {
    money: number; // 该工作的报酬
    hard: number; // 该工作的难度
    constructor(money: number, hard: number) {
        this.money = money;
        this.hard = hard;
    }
}
```

给定一个Job类型的数组jobarr，表示所有的工作。给定一个int类型的数组arr，表示所有小伙伴的能力。

返回int类型的数组，表示每一个小伙伴按照牛牛的标准选工作后所能获得的报酬。

思路（有序表）：

1. 将jobarr以难度从小到大排序，如果相同难度以报酬从大到小排序；
2. 相同难度只保留报酬最多的job，如果难度递增，报酬减少的job也要删除。
3. 这样只要选择小于能力的最大难度的工作为最合适的；

```
function ChooseWork(job: Job[], ability: number): Job {
    const ableJob = job.filter(item => item.hard <= ability);
    ableJob.sort((a, b) => a.money - b.money);
    return ableJob[ableJob.length - 1];
}
```

javascript实现有序表?