解题技巧和思想：

- 窗口不回退模型
- 打标
- 预处理

题目一（窗口不回退）

给定一个有序数组 arr，代表数轴上从左到右有 n 个点 arr[0]、arr[1]...arr[n - 1]，
给定一个正数 L，代表一根长度为 L 的绳子，求绳子最多能覆盖其中的几个点。

思路 1：

1. 遍历 arr，假设当前i，寻找 arr 中大于arr[i]-L最左的位置index，此时绳子覆盖了index-i个点。遍历完成后返回最大的值；
2. 采用二分法寻找 arr 中大于 value 的最左位置；
3. 时间复杂度 O(n*logn)

```
function coverMaxPoint(arr, L) {
    let res = 1;
    for (let i = 0; i < arr.length; i++) {
        //
        let index = nearestIndex(arr, i, arr[i] - L);
        res = res > i - index + 1 ? res : i - index + 1;
    }
    return res;
}

// 在arr[0..R]范围上，找满足>=value的最左位置
function nearestIndex(arr, R, value) {
    let left = 0;
    let index = R;

    while (left < R) {
        let mid = left + ((R - left) >> 1);
        if (arr[mid] >= value) {
            index = mid;
            R = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return index;
}
```

思路二：

1. 生成不后退的滑动窗口，每次返回覆盖的点数
2. left 开始指向 arr[0]，right 向右移动，但需要保证窗口长度不能大于 L；如果当 right 指向 arr[i+1]时窗口长度超过 L，left 向右移动；
3. 时间复杂度 O(n)

```
function coverMaxPoint2(arr, L) {
    let left = 0;
    let right = 0;
    let res = 0;
    while (left < arr.length) {
        // 窗口在L范围内，right++
        if (right < arr.length && arr[right + 1] - arr[left] <= L) {
            right++;
        } else {
            // 窗口要超过L，left++
            res = Math.max(res, right - left + 1);
            left++;
        }
    }
    return res;
}
```

题目二（打表）

小虎去附近的商店买苹果，奸诈的商贩使用了捆绑交易，只提供 6 个每袋和 8 个每袋的包装包装不可拆分。可是小虎现在只想购买恰好 n 个苹果，小虎想购买尽量少的袋数方便携带。如果不能购买恰好 n 个苹果，小虎将不会购买。输入一个整数 n，表示小虎想购买的个苹果，返回最小使用多少袋子。如果无论如何都不能正好装下，返回-1。

思路：

1. 首先全部的苹果用容量为8的袋子装（i个8容量袋子），如果不满足条件i-1，剩余的用容量为6的袋子装...
2. 当最后无法装下去的苹果数量大于24（8和6的最小公倍数，此时可以分解为3*8+剩余的，不满足最小使用袋子数）时，返回-1；

```
function appleMinBags(n) {
    if (n <=0) {
        return -1;
    }
    let rest = n % 8;
    let count8 = Math.floor(n / 8);
    while (count8 >= 0 && rest < 24) {
        if (rest % 6 === 0) {
            return count8 + rest / 6
        } else {
            count8--;
            rest = n - count8 * 8;
        }
    }
    return -1;
}
```

思路：

1. 输入整数输出整数，可以使用打表发。使用对数器打印答案，寻找规律。
2. 奇数返回-1，6 8 返回1，12 14 16 返回2，之后18~25返回3，26~33返回4.... (n-18)/8 + 3

```
function appleMinBags2(n) {
    // 奇数为-1
    if (n % 2 !==0) {
        return -1;
    }
    if (n < 18) {
        return n === 6 || n === 8 ? 1 : (n === 12 || n === 14 || n === 16 ? 2 : -1)
    }

    return parseInt((n - 18) / 8) + 3;
}
```

题目二（打表扩展）

牛牛和羊羊都很喜欢青草。今天他们决定玩青草游戏。

最初有一个装有n份青草的箱子,牛牛和羊羊依次进行,牛牛先开始。在每个回合中,每个玩家必须吃一些箱子中的青草,所吃的青草份数必须是4的x次幂,比如1,4,16,64等等。
不能在箱子中吃到有效份数青草的玩家落败。

假定牛牛和羊羊都是按照最佳方法进行游戏,请输出胜利者的名字。

举例： 2份草，先1后1，后赢； 4份草 先4，先赢。。。

思路:

1. 甲先手依次尝试拿1份、4份、8份...，则如果乙后手的winner(n-base)为后手赢（乙的winner过程中，乙为先手，甲为后手，所以如果winner(n-base)返回后手赢则代表甲会赢）

```
function winner(n) {
    // 0  1  2  3  4
    // 后 先 后 先 先
    if (n < 5) {
        return n == 0 || n == 2 ? "后手" : "先手";
    }
    //n >= 5时
    let base = 1; //先手决定吃的草
    //有问题
    while (base <= n) {
        //当前一共n份草，先手吃掉的是base份，n-base是留给后手的草
        //母过程 先手 在子过程是后手
        if (winner(n - base) === "后手") {
            return "先手";
        }
        if (base > n / 4) {
            // 防止base*4之后溢出
            break;
        }
        base *= 4;
    }
    return "后手";
}
```

打表

0 后手 1 先手 2 后手 3 先手 4 先手 5 后手 6 先手 7 后手 8 先手
9 先手 10 后手 11 先手 12 后手 13 先手 14 先手 15 后手 16 先手
17 后手 18 先手 19 先手 20 后手 21 先手 22 后手 23 先手 24 先手 25 后手 26 先手 27 后手 28 先手 29 先手 30 后手 31 先手 32 后手 33 先手 34 先手 35 后手 36 先手 37 后手 38 先手 39 先手 40 后手 41 先手 42 后手 43 先手 44 先手 45 后手 46 先手 47 后手 48 先手 49 先手 50 后手 51 先手 52 后手 53 先手 54 先手 55 后手 56 先手 57 后手 58 先手 59 先手 60 后手 61 先手 62 后手 63 先手 64 先手 65 后手 66 先手 67 后手 68 先手 69 先手 70 后手 71 先手 72 后手 73 先手 74 先手 75 后手 76 先手 77 后手 78 先手 79 先手

可发现：0 2 5 7 10 12 15 17 20 22 ...为后手（n % 5 === 0 || n % 5 === 2）

题目三（预处理数据）

牛牛有一些排成一行的正方形。每个正方形已经被染成红色或者绿色。牛牛现在可以选择任意一个正方形然后用这两种颜色的任意一种进行染色,这个正方形的颜色将会被覆盖。牛牛的目标是在完成染色之后,每个红色 R 都比每个绿色 G 距离最左侧近。牛牛想知道他最少需要涂染几个正方形。

如样例所示: s = RGRGR。我们涂染之后变成 RRRGG 满足要求了,涂染的个数为 2,没有比这个更好的涂染方案。

思路1：

1. 将字符串转成数组；
2. 遍历数组，以当前下标i为分界，分成左边和右边；
3. 左边的全部染成R，右边的染成G。遍历0~i-1统计有多少G需要染成R，i~n有多少R需要染成G，两者相加则是以当前下标i为界的结果；
4. 遍历完成返回最小值；

```
function colorLeftRight(s) {
    // 字符串转成数组
    const arr = s.split("");
    let res = Infinity;
    for (let i = 0; i < arr.length; i++) {
        let temp = 0;
        // 0~i-1统计有多少G需要染成R
        for (let left = 0; left < i; left++) {
            if (arr[left] === "G") {
                temp++;
            }
        }
        // i~n有多少R需要染成G
        for (let right = i; right < arr.length; right++) {
            if (arr[right] === "R") {
                temp++;
            }
        }
        // 返回较小值
        res = Math.min(temp, res);
    }
    return res;
}
```

思路2：（预处理）

1. 生成两个数组分别记录以当前下标i为界左边有多少个G，右边有多少个R；
2. 统计有多少G需要染成R（R需要染成G）直接查找数组，减少遍历的时间复杂度；

```
function colorLeftRight2(s) {
```

```
    const arr = s.split("");
    let res = Infinity;
    // 记录0~i范围内有多少个R
    let leftGList = [];
    let rightRList = [];
    // 包括i
    leftGList[0] = arr[0] === "G" ? 1 : 0;
    // 不包括i
    rightRList[arr.length - 1] = 0;
    for (let i = 1; i < arr.length; i++) {
        leftGList[i] = arr[i] === "G" ? leftGList[i - 1] + 1 : leftGList[i - 1];
    }
    for (let i = arr.length - 2; i >= 0; i--) {
        rightRList[i] =
            arr[i + 1] === "R" ? rightRList[i + 1] + 1 : rightRList[i + 1];
    }
    for (let i = 0; i < arr.length - 1; i++) {
        let temp = leftGList[i] + rightRList[i];
        res = Math.min(temp, res);
    }
    return res;
}
```

题目四（预处理）

给定一个 N*N 的矩阵 matrix，只有 0 和 1 两种值，返回边框全是 1 的最大正方形的边长长度。
例如:

```
01111
01001
01001
01111
01011
```

其中边框全是 1 的最大正方形的大小为 4*4，所以返回 4。

思路1：

1. 遍历整个矩阵，假设当前位置为（i，j）；
2. 遍历所有可能的border形成以当前位置为左上角点的正方形；
3. 遍历正方形的四条边，判断是否都是1；
4. 返回最大的border；
5. 时间复杂度O(n^4)

```
function maxOneBorderSize(matrix) {
    let res = 0;
    let N = matrix.length;
    // 遍历矩阵
    for (let row = 0; row < N; row++) {
        for (let col = 0; col < N; col++) {
            // 遍历可能的border
            for (let border = 1; row + border - 1 < N && col + border - 1 < N; border++) {
                let flag = true;
                // 上
                for (let i = col; i <= col + border - 1; i++) {
```

```
                    if (matrix[row][i] === 0) {
                        flag = false
                        break
                    }
                }
                // 下
                for (let i = col; i <= col + border - 1; i++) {
                    if (matrix[row+border-1][i] === 0) {
                        flag = false
                        break
                    }
                }
                // 左
                for (let i = row; i <= row + border - 1; i++) {
                    if (matrix[i][col] === 0) {
                        flag = false
                        break
                    }
                }
                // 右
                for (let i = row; i <= row + border - 1; i++) {
                    if (matrix[i][col+border-1] === 0) {
                        flag = false
                        break
                    }
                }
                if (flag) {
                    res = Math.max(res, border)
                }
            }
        }
    }
    return res;
}
```

思路2： （预处理）

1. 用两个N*N的矩阵right和down分别记录从当前位置（i，j）往右和往下所连续1的个数（包含i和j位置）；
2. 判断正方形边是否都是1时，从顶点处获取right和down中值，判断是否大于等于border，如果是则表明边上都是1；
3. 时间复杂度O(n^3)

```
function maxOneBorderSize2(matrix) {
    let res = 0;
    const N = matrix.length;
    const right = new Array(N);
    for (let i = 0; i < N; i++) {
        right[i] = new Array(N);
        for (let j = N - 1; j >= 0; j--) {
            if (j === N - 1) {
                right[i][j] = matrix[i][j]
            } else {
                right[i][j] = matrix[i][j] === 0 ? 0 : matrix[i][j] + right[i][j+1]
            }
        }
    }
    const down = new Array(N);
    for (let i = 0; i < N; i++) {
        down[i] = new Array(N);
    }
```

```
    for (let j = 0; j < N; j++) {
        for (let i = N - 1; i >= 0; i--) {
            if (i === N - 1) {
                down[i][j] = matrix[i][j]
            } else {
                down[i][j] = matrix[i][j] === 0 ? 0 : matrix[i][j] + down[i+1][j]
            }
        }
    }

    // 遍历矩阵
    for (let row = 0; row < N; row++) {
        for (let col = 0; col < N; col++) {
            // 遍历border
            for (let border = 1; row + border - 1 < N && col + border - 1 < N; border++) {
                // 判断边是否为1
                if (
                    right[row][col] < border ||
                    right[row + border - 1][col] < border ||
                    down[row][col] < border ||
                    down[row][col + border - 1] < border) {
                    continue;
                }
                res = Math.max(res, border);
            }
        }
    }
    return res;
}
```

题目五

给定一个函数 f，可以 1 ~ 5 的数字等概率返回一个。请加工出 1 ~ 7 的数字等概率返回一个的函数 g。

思路:

1. 将f函数加工为等概率返回0 1的函数r;
2. 用三位二进制表示返回结果，每一位上的0 1用函数r决定，如果结果为7从头重新决定;

```
function f() {
    return Math.floor(Math.random() * 5) + 1;
}

function rand5ToRand7(f) {
    // 将f函数改成0 1发生器
    function r() {
        let res = 0;
        do {
            res = f();
        } while (res === 3)
        return res < 3 ? 0 : 1;
    }
    function g() {
        let res = 0;
        do {
            res = (r() << 2) + (r() << 1) + r()
        } while (res === 7)
```

```
            return res + 1;
        }
        return g;
    }
```

给定一个函数 f，可以 a ~ b 的数字等概率返回一个。请加工出 c ~ d 的数字等概率返回一个的函数 g。

思路：

1. 将函数f加工成等概率生成0 1的函数r，；
2.

给定一个函数 f，以 p 概率返回 0，以 1-p 概率返回 1。请加工出等概率返回 0 和 1 的函数 g

思路：

1. 用两位二进制进行辅助，每一位分别用f确定；
2. 00概率p^2，11概率（1-p）^2，01和10等概率p(1-p);
3. 01返回0，10返回1，其他值重新确定；

题目六

给定一个非负整数 n，代表二叉树的节点个数。返回能形成多少种不同的二叉树结构

思路1：

1. 遍历左树可能的节点个数i(0~n-1)，总个数为process(i)*process(n-i-1) 左树结构数×右树结构数

```
function uniqueBST(n) {
    function process(n) {
        if (n < 0) {
            return 0;
        }
        if (n === 0 || n === 1) {
            return 1;
        }
        if (n === 2) {
            return 2;
        }
        let res = 0;
        for (let i = 0; i < n; i++) {
            let leftWays = process(i);
            let rightWays =  process(n-i-1);
            res += leftWays * rightWays
        }
        return res;
    }
    return process(n)
}
```

动态规划：

```
// 动态规划
function uniqueBST2(n) {
    const dp = new Array(n+1).fill(0);
    dp[0] = 1;
    for (let i = 1; i <= n; i++) {
        // 左子树的节点个数 0 ~ i- 1
        for (let j = 0; j < i; j++) {
            dp[i] += dp[j] * dp[i-j-1]
        }
    }
    return dp[n]
}
```

题目七

一个完整的括号字符串定义规则如下:

1. 空字符串是完整的。
2. 如果 s 是完整的字符串，那么(s)也是完整的。
3. 如果 s 和 t 是完整的字符串，将它们连接起来形成的 st 也是完整的。

例如，"(()())", ""和"(())()"是完整的括号字符串，"())(", "()(" 和 ")" 是不完整的括号字符串。

牛牛有一个括号字符串 s,现在需要在其中任意位置尽量少地添加括号,将其转化
为一个完整的括号字符串。请问牛牛至少需要添加多少个括号。

思路:

1. 将字符串转化为数组，用变量count辅助记录，遍历数组，如果当前为“(” count加一，如果为“)” count减一；
2. 如果遍历过程中count变为-1，则说明需要在此下标前加入“(”配对，res加一，并将count重置为0；
3. 如果遍历结束count大于0，则说明需要在结尾加入count个“)”，res加count；

```
function needParentheses(str) {
    let arr = str.split('');
    let count = 0;
    let res = 0;
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === "(") {
            count++
        } else {
            count--
        }
        if (count < 0) {
            res++
            count = 0
        }
    }
    return count + res;
}
```