【第四十周】本月月度测试题

1、2100. 适合打劫银行的日子

1. 前缀和思想
2. 第一个数组pre[i]：pre记录从security[i]开始，左边连续非递增的元素数目，
3. 第二个数组suff[i]：suff记录从security[i]开始，右边连续非递减的元素数目。
4. 只有符合pre[i] >= time && suff[i] >= time的security[i]才适合。

```
 /**
 * @param {number[]} security
 * @param {number} time
 * @return {number[]}
 */
var goodDaysToRobBank = function(security, time) {
    const n = security.length;
    //--------前缀
    const pre = new Array(n).fill(0);
    for (let i = 1; i < n; i ++){
        if (security[i - 1] >= security[i]){
            pre[i] = pre[i - 1] + 1;
        }
    }
    //--------后缀
    const suff = new Array(n).fill(0);
    for (let i = n - 2; i > -1; i --){
        if (security[i] <= security[i + 1]){
            suff[i] = suff[i + 1] + 1;
        }
    }
    //--------结果
    let res = new Array(0);
    for (let i = time; i < n - time; i ++){
        if (pre[i] >= time && suff[i] >= time){
            res.push(i);
        }
    }
    return res;

};
```

2、53. 最大子数组和

1. 动态规划的是首先对数组进行遍历，当前最大连续子序列和为 sum，结果为 ans
2. 如果 sum > 0，则说明 sum 对结果有增益效果，则 sum 保留并加上当前遍历数字
3. 如果 sum <= 0，则说明 sum 对结果无增益效果，需要舍弃，则 sum 直接更新为当前遍历数字
4. 每次比较 sum 和 ans的大小，将最大值置为ans，遍历结束返回结果
5. 时间复杂度：O(n)

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var maxSubArray = function(nums) {
    let ans = nums[0];
    let sum = 0;
    for(const num of nums) {
        if(sum > 0) {
            sum += num;
        } else {
            sum = num;
        }
        ans = Math.max(ans, sum);
    }
    return ans;
};
```

3、LCP 30. 魔塔游戏

1. 首先，如果总和加上初始血量不为正数，说明无论如何调整顺序，都无法通过，直接返回 -1
2. 其次，由于要求最小调整次数，所以我们优先将降低血量最多的位置调整到最后，
3. 这就需要动态记录当前遍历过所有位置中负血量的最小值（绝对值最大值），所以利用优先队列存储所有遍历到的负数
4. 遍历到的正数（提升血量）并不需要记录到优先队列中，因为不会降低血量

```
var magicTower = function(nums) {
    let hp = 1;
    hp += nums.reduce((a,b) => a + b);
    if(hp <= 0) return -1;
    hp = 1;
    let damage = [];
    let last = 0;
    for (let i = 0; i < nums.length; i++) {
```

```
        if (nums[i] < 0) {
            damage.push(nums[i]);
            if (hp + nums[i] <= 0) {//这回合过后就要死了，需要把前面扣最多的血移到最后去
                last++;
                let min = Math.min(...damage);
                damage.splice(damage.indexOf(min), 1);
                hp -= min;
            }
        }
        hp += nums[i];
    }
    return last;
};
```

4、877. 石子游戏

1. 定义二维数组 dp，其行数和列数都等于石子的堆数， dp[i][j] 表示当剩下的石子堆为下标 i 到下标 j 时，即在下标范围 [i, j]中，当前玩家与另一个玩家的石子数量之差的最大值，注意当前玩家不一定是先手 Alice。
2. 只有当 i ≤ j 时，剩下的石子堆才有意义，因此当 i > j 时，dp[i][j] = 0。
3. 当 i== j 时，只剩下一堆石子，当前玩家只能取走这堆石子，因此对于所有 0 ≤ i < nums.length，都有dp[i][i] = piles[i]。
4. 当 i < j时，当前玩家可以选择取走 piles[i] 或 piles[j]，然后轮到另一个玩家在剩下的石子堆中取走石子。在两种方案中，当前玩家会选择最优的方案，使得自己的石子数量最大化。因此可以得到如下状态转移方程：dp[i][j] =max(*piles*[*i*]−*dp*[*i*+1][*j*],*piles*[*j*]−*dp*[*i*][*j*−1])
5. 最后判断 dp[0][piles.length - 1] 的值，如果大于 0，则 Alice 的石子数量大于 Bob 的石子数量，因此 Alice 赢得比赛，否则 Bob 赢得比赛。
6. dp[i][j] 的值只和 dp[i+1][j] 与 dp[i][j−1] 有关，就是在计算 dp 的第 i 行的值时，只需要使用到 dp 的第 i 行和第 i+1 行的值，因此可以使用一维数组代替二维数组，对空间进行优化。

```
var stoneGame = function(piles) {
    const length = piles.length;
    const dp = new Array(length).fill(0);
    for (let i = 0; i < length; i++) {
        dp[i] = piles[i];
    }
    for (let i = length - 2; i >= 0; i--) {
        for (let j = i + 1; j < length; j++) {
            dp[j] = Math.max(piles[i] - dp[j], piles[j] - dp[j - 1]);
        }
    }
    return dp[length - 1] > 0;
};
```

5、623. 在二叉树中增加一行

1. 广度优先搜索是最容易理解且最直观的一种方法。
2. 我们将根节点放入队列 queue。
3. 在每一轮搜索中，如果 queue 中节点的深度为 d - 1（显然 queue 中所有的节点都在同一深度），
4. 我们就退出搜索，并为 queue 中所有节点添加新的子节点；
5. 否则我们将 queue 中所有节点的子节点放入新的队列 temp 中，再用 temp 替代 queue。

```
var addOneRow = function(root, v, d) {
    if (d === 1) {
        var newRoot = new TreeNode(v);
        newRoot.left = root;
        return newRoot;
    }
    let que = []; que.push(root);
    // que的第0层是d=1， 需要在d-1停止，因此循环到d-2
    for (let i = 0;i < d-2; i++)
        for (let j = que.length; j; j--) {
            var t = que.shift();
            if (t.left) que.push(t.left);
            if (t.right) que.push(t.right);
        }
    // 对当前d-1层的所有node接上新的node， 再在新的node下面接上旧的下一层的node
    while (que.length) {
        var t = que.shift();
        var newL = new TreeNode(v), newR = new TreeNode(v);
        // 在新的node下面接上旧的node
        newL.left = t.left, newR.right = t.right;
        t.left = newL, t.right = newR;
    }
    return root;
};
```

6、934. 最短的桥

1. 为区分两个岛屿，将其中一个岛屿使用深度优先遍历dfs进行染色
2. 扩张其中一个岛屿边缘，直到与另一个岛屿相连，记录扩张次数，即为桥梁长度

```
/**
 * @param {number[][]} A
 * @return {number}
 */
```

```
var shortestBridge = function(A) {
  const rows = A.length;
  const cols = A[0].length;
  const queue = [];

  for (let i = 0; i < rows; i++) {
    if(A[i].includes(1)) {
      dfs(i, A[i].indexOf(1));
      break;
    }
  }

  function dfs(x, y) {
    if (x < 0 || x >= rows || y < 0 || y >= cols || A[x][y] === 2) {
      return;
    }
    if (A[x][y] === 0) {
      queue.push([x, y]); // 存储岛屿边缘
      return;
    }
    A[x][y] = 2; // 染色
    dfs(x - 1, y);
    dfs(x + 1, y);
    dfs(x, y - 1);
    dfs(x, y + 1);
  }

  let curQueue = [];
  let result = 0;
  while (queue.length) {
    const [x, y] = queue.shift();
    if (x < 0 || x >= rows || y < 0 || y >= cols || A[x][y] === 2) {
      updateLen();
      continue;
    }
    if (A[x][y] === 1) { // 扩张到另一岛屿时，桥梁建成
      break;
    }
    if (A[x][y] === 0) { // 存储下一轮扩张的坐标
      curQueue.push([x - 1, y], [x + 1, y], [x, y - 1], [x, y + 1]);
    }
    A[x][y] = 2; // 标记已扩张部分，防止重复访问
    updateLen();
  }

  function updateLen() {
    if (!queue.length) { // 一轮扩张结束时，更新桥梁长度及下一轮扩张坐标
      result++;
      queue.push(...curQueue);
```

```
            curQueue = [];
        }
    }
    return result;
};
```

7、剑指 Offer 14- II. 剪绳子 II

1. 贪心思路
2. 绳子长度大于4的时候，不断减去长度3
3. 直到绳子长度小于等于4
4. 最后所有段相乘，就是最大的乘积

```
/**
 * @param {number} n
 * @return {number}
 */
var cuttingRope = function(n) {
    // 特殊情况处理
    const arr = [null, null, 1, 2, 4];
    if (n <= 4) return arr[n];
    const mod = 1000000007;
    let res = 1;
    while (n > 4) {
        // 每次减掉3
        res = (res * 3) % mod;
        n -= 3;
    }
    // 最后剩下一段小于等于4的长度
    res *= n;
    return res % mod;
};
```

8、1863. 找出所有子集的异或总和再求和

1. 题目要求我们枚举所有子集的异或和，我们用二进制枚举的方式来进行（比写DFS快很多），
2. 用一个长度为n的二进制数来表示所有的子集，每一位对应nums 每一位取还是不取（二进制1和0）
3. 比如：1010 对应的就是nums数组的第一个数和第三个数组成的子集。
4. 这个二进制数从0开始（空子集）到1<<n 结束（每一个数都取，全集），我们再枚举这每个二进制数的每一位，如果这一位为1，则取出nums数组的对应位的数进行异或，最后把每个二进制数表示的子集的异或结果相加即可。

```
/**
* @param {number[]} nums
* @return {number}
*/
var subsetXORSum = function(nums) {
        let  n = nums.length;
        let  ans = 0;
        for (let  i = 0; i < (1 << n); ++i) {
            let  res = 0;
            for (let  j = 0; j < n; ++j) {
                if (i & (1 << j)) {
                    res = res ^ nums[j];
                }
            }
            ans += res;
        }
        return ans;
    }
```

9、1094. 拼车

1. 题目意思，你是公交车司机，公交车的最大载客量为capacity，沿途要经过若干车站，给你一份乘客行程表trips
2. 其中trip[i]=[num,start,end]代表着有num个旅客要从站点start上车，到站点end下车，让你计算是否能够把所有旅客一次性运送完毕
3. 我们使用差分数组的技巧，trips[i]代表着一组区间操作，旅客的上车和下车就相当于数组的区间加减，只要结果数组中的元素都小于capacity，就说明可以不超载运输所有旅客

```
function Difference(nums) {
 // 差分数组
 this.diff = new Array(nums.length).fill(0);
 // 根据初始数组构建差分数组
 for (let i = 1; i < nums.length; i++) {
   this.diff[i] = nums[i] - nums[i - 1];
 }
 // 给闭区间[i,j]增加val （可以是负数）
 this.increment = function (i, j, val) {
   this.diff[i] += val;
   if (j + 1 < this.diff.length) {
     this.diff[j + 1] -= val;
   }
 };
 this.result = function () {
```

```
    let res = new Array(this.diff.length).fill(0);
    res[0] = this.diff[0];
    for (let i = 1; i < this.diff.length; i++) {
      res[i] = res[i - 1] + this.diff[i];
    }
    return res;
  };
}
/**
 * @param {number[][]} trips
 * @param {number} capacity
 * @return {boolean}
 */
var carPooling = function (trips, capacity) {
  // 最多有1000个车站
  let nums = new Array(1001).fill(0);
  // 构建差分数组
  let df = new Difference(nums);
  for (let trip of trips) {
    // 乘客数量
    let val = trip[0];
    // 第trip[1]站乘客上车
    let i = trip[1];
    // 第trip[2]站乘客已经下车，即乘客在车上的区间是[trip[1],trip[2-1]]
    let j = trip[2] - 1;
    df.increment(i, j, val);
  }
  let res = df.result();
  for (let i = 0; i < res.length; i++) {
    if (capacity < res[i]) {
      return false;
    }
  }
  return true;
};
```

10、687. 最长同值路径

1. 第12课讲过的题 递归（DFS）思路
2. 递归函数是求一个子树可以向父节点提供的路径长度
3. 对于当前节点，左子树能提供的长度为 left，如果当前节点值等于左子节点的值，则左链的长度等于left+1，否则为0
4. 对于当前节点，右子树能提供的长度为right，如果当前节点值等于右子节点的值，则右链的长度等于right+1，否则为0
5. 当前子树对父节点提供的最大长度为左右链中较大的一个

6. 当前子树的左右链之和，去和全局最大值比较，试图更新它

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var longestUnivaluePath = function(root) {
    let res = 0

    const dfs = (root) => {
        if (root == null) {
            return 0
        }
        const left = dfs(root.left)
        const right = dfs(root.right)

        let leftPath = 0, rightPath = 0

        if (root.left && root.left.val == root.val) {
            leftPath = left + 1
        }
        if (root.right && root.right.val == root.val) {
            rightPath = right + 1
        }
        res = Math.max(res, leftPath + rightPath)

        return Math.max(rightPath, leftPath)
    }

    dfs(root)
    return res
}
```