【第三十二周】十月月度测试题

1、696. 计数二进制子串

1. 我们可以将字符串 s 按照 0 和 1 的连续段分组，存在counts 数组中，例如 s=00111011，可以得到这样的 counts 数组：counts={2,3,1,2}。
2. 这里 counts 数组中两个相邻的数一定代表的是两种不同的字符。假设 counts 数组中两个相邻的数字为 uu 或者 ，它们对应着 个 和 个 1，或者 u 个 1 和 v 个 0。它们能组成的满足条件的子串数目为 min{u,v}，即一对相邻的数字对答案的贡献。
3. 我们只要遍历所有相邻的数对，求它们的贡献总和，即可得到答案。

```
var countBinarySubstrings = function(s) {
    const counts = [];
    let ptr = 0, n = s.length;
    while (ptr < n) {
        const c = s.charAt(ptr);
        let count = 0;
        while (ptr < n && s.charAt(ptr) === c) {
            ++ptr;
            ++count;
        }
        counts.push(count);
    }
    let ans = 0;
    for (let i = 1; i < counts.length; ++i) {
        ans += Math.min(counts[i], counts[i - 1]);
    }
    return ans;
};
```

2、686. 重复叠加字符串匹配

1. a循环多次后包含字符串b,
2. 用b的长度除a的长度，向上取整得minLen 到 minLen + 1个a并在一起，
3. 设这个合并值为c，那只要c包含b就成立了

```
/**
 * @param {string} a
 * @param {string} b
 * @return {number}
 */
var repeatedStringMatch = function(a, b) {
    const aLen = a.length;
    const bLen = b.length;
    let maxLen = Math.ceil(bLen / aLen) + 1;
    let c = '';
```

```
    for(let i = 0; i < maxLen; i++) {
        c += a;
        if(i > maxLen - 3) {
            if(c.includes(b)) return i + 1;
        }
    }
    return -1;
};
```

3、820. 单词的压缩编码

1. 剔除重复词尾的思路，通过哈希表降低查询的复杂度(空间换时间)。
2. 对 words 中的每个元素的词尾做切片并比对，如果词尾出现在 words 中，则删除该元素。

```
/**
 * @param {string[]} words
 * @return {number}
 */
var minimumLengthEncoding = function (words) {
    let hashSet = new Set(words);
    for (let item of hashSet) {
        for (let i = 1; i < item.length; i++) {
            // 切片，看看是否词尾在 hashSet 中，切片从1开始，只看每个单词的词尾
            let target = item.slice(i);
            hashSet.has(target) && hashSet.delete(target);
        }
    }
    let result = 0;
    // 根据 hashSet 中剩余元素计算最终长度
    hashSet.forEach(item => result += item.length + 1)
    return result
};
```

4、409. 最长回文串

1. 首先用map统计每个字母出现的次数
2. 再遍历map，res累加上(累加上 出现次数-次数对2取模)
3. 比如一个字母出现4次，那么就能为最后答案多贡献4个。如果出现3次，只能贡献2个。出现1次，没有贡献。
   最后判断是否有出现奇数次数的字母，判断方法是res是否小于字符串长度，只要有奇数次数的字母，res < len
4. 有奇数次数的字母，则res++，因为多1个可以放到最中间，也是回文串

```
const longestPalindrome = s => {
    // 统计各个字母出现的次数
    const map = new Map();
    const len = s.length;
    for (let i = 0; i < len; i++) {
        map.set(s[i], (map.get(s[i]) || 0) + 1);
    }
```

```
    let res = 0;
    // 遍历map
    for (const item of map) {
        // res累加上 出现次数-次数对2取模
        res += item[1] - (item[1] % 2);
    }
    // 如果有奇数字母的，res加1
    if (res < len) res++;

    return res;
};
```

5、647. 回文子串

1. 方法：动态规划
2. dp[i][j]：表示区间范围[i,j] （注意是左闭右闭）的子串是否是回文子串，如果是dp[i][j]为true，否则为false
3. 若s[i]与s[j]不相等，dp[i][j]=false
4. 若s[i]与s[j]相等，（1）区间长度<=2，dp[i][j]=true；（2）区间长度>2，需要判断里面一层dp[i+1][j-1]
5. dp数组初始化：全部初始化为false
6. 遍历时，用一个变量res记录true的个数
7. 因为dp[i][j]依赖于dp[i+1][j-1]，所以左下角开始遍历

```
const countSubstrings = s => {
    const len = s.length;
    let res = 0;
    // 创建dp数组并初始化为false
    const dp = new Array(len).fill(0).map(x => new Array(len).fill(false));

    for (let i = len - 1; i >= 0; i--) {
        for (let j = i; j <= len - 1; j++) {
            // s[i] === s[j]的情况下才可能出现回文串
            if (s[i] === s[j]) {
                if (j - i <= 1) {
                    // 区间长度1或2，如：
                    // 'a'或'aa'，肯定是回文串
                    res++;
                    dp[i][j] = true;
                } else {
                    // 区间长度大于2，需要判断里面一层
                    if (dp[i + 1][j - 1]) {
                        res++;
                        dp[i][j] = true;
                    }
                }
            }
        }
    }
    return res;
};
```

6、205. 同构字符串

1. 方法：哈希表
2. 需要我们判断 s 和 t 每个位置上的字符是否都一一对应，即 s 的任意一个字符被 t 中唯一的字符对应，同时 t 的任意一个字符被 s 中唯一的字符对应。这也被称为「双射」的关系。
3. 以示例 2 为例，t 中的字符 a 和 r 虽然有唯一的映射 o，但对于 s 中的字符 o 来说其存在两个映射 {a,r}，故不满足条件。
4. 因此，我们维护两张哈希表，第一张哈希表 s2t 以 s 中字符为键，映射至 t 的字符为值，第二张哈希表t2s 以 t 中字符为键，映射至 s 的字符为值。
5. 从左至右遍历两个字符串的字符，不断更新两张哈希表，如果出现冲突（即当前下标 index 对应的字符 s[index] 已经存在映射且不为 t[index] 或当前下标 index 对应的字符 t[index] 已经存在映射且不为s[index]）时说明两个字符串无法构成同构，返回 false。
6. 如果遍历结束没有出现冲突，则表明两个字符串是同构的，返回 true 即可。

```
var isIsomorphic = function(s, t) {
    const s2t = {};
    const t2s = {};
    const len = s.length;
    for (let i = 0; i < len; ++i) {
        const x = s[i], y = t[i];
        if ((s2t[x] && s2t[x] !== y) || (t2s[y] && t2s[y] !== x)) {
            return false;
        }
        s2t[x] = y;
        t2s[y] = x;
    }
    return true;
};
```

7、211. 添加与搜索单词 - 数据结构设计

1. 方法：字典树
2. 查找单词时，深度优先单词树中是否有一条路，每个字符都对应，并且标记过 `isEnd=true`。其中，`'.'`可以代表任意字符。

```
// 字典树
class TrieNode {
    constructor() {
        this.children = new Array(26).fill(0);
        this.isEnd = false;
    }
    insert(word) {
        let node = this;
        for (let i = 0; i < word.length; i++) {
            const ch = word[i];
            // 计算当前字符的索引
            const index = ch.charCodeAt() - 'a'.charCodeAt();
```

```
            // 若索引位置还没有单词，则在此新建字典树
            if (node.children[index] === 0) node.children[index] = new
TrieNode();
            node = node.children[index];
        }
        // 结尾标记单词结束了
        node.isEnd = true;
    }
    getChildren() {
        return this.children;
    }
    isEnd() {
        return this.isEnd;
    }
}

class WordDictionary {
    constructor() {
        this.trieRoot = new TrieNode();
    }
    addWord(word) {
        this.trieRoot.insert(word);
    }
    search(word) {
        const dfs = (index, node) => {
            // 若索引长度等于单词数，说明遍历完了，返回isEnd
            if (index === word.length) return node.isEnd;

            const ch = word[index];
            if (ch !== '.') {
                // 当前字符是字母，必须保证一致
                const child = node.children[ch.charCodeAt() - 'a'.charCodeAt()];
                if (child && dfs(index + 1, child)) return true;
            } else {
                // 当前字符是点，点可以代表任何字符
                // 只要有一个子节点是true即可
                for (const child of node.children) {
                    if (child && dfs(index + 1, child)) return true;
                }
            }
            // 字符不存在，返回false
            return false;
        };
        return dfs(0, this.trieRoot);
    }
}
```

8、745. 前缀和后缀搜索

1. 方法：后缀树+前缀树
2. 往Trie树插入单词时，单词本身包含所有的前缀组合。那么考虑将所有的后缀组合提前，再加一个特殊字符，组成 后缀+特殊字符+单词 的结构，如 suffix+'#'+word 。那么调用f() 时，只需要查询 suffix+'#'+word 的权重即可。

3. 对于 apple 这个单词，我们可以在单词查找树插入每个后缀，后跟 “#” 和单词。
例如，我们将在单词查找树中插入 '#apple', 'e#apple', 'le#apple', 'ple#apple', 'pple#apple', 'apple#apple'。然后对于 prefix = "ap", suffix = "le" 这样的查询，我们可以通过查询单词查找树找到 le#ap

```
 function TrieNode() {
  this.next = new Map();
  this.weight = 0; //权重，对应单词下标
}
function Trie() {
  this.root = new TrieNode();
}
Trie.prototype.insert = function (word, weight) {
  if (!word) return;
  let node = this.root;
  for (let c of word) {
    if (!node.next.has(c)) {
      node.next.set(c, new TrieNode());
    }
    node = node.next.get(c);
    node.weight = weight; // 更新weight
  }
};
var WordFilter = function (words) {
  this.tree = new Trie();
  for (let weight = 0; weight < words.length; weight++) {
    let word = words[weight];
    let suffix = '';
    for (let i = word.length; i >= 0; i--) {
      suffix = word.slice(i, word.length);
      this.tree.insert(suffix + '#' + word, weight);
    }
  }
};

/**
 * @param {string} prefix
 * @param {string} suffix
 * @return {number}
 */
WordFilter.prototype.f = function (prefix, suffix) {
  let target = suffix + '#' + prefix;
  let node = this.tree.root;
  for (let c of target) {
    if (!node.next.has(c)) return -1;
    node = node.next.get(c);
  }
  return node.weight;
};
```

9、1161. 最大层内元素和

1. 方法：DFS：递归实现的中序遍历

2. 递归实现的中序遍历非常简单：遵循 `Left->Node->Right` 的顺序。例如：首先递归访问 左孩子，然后访问父节点，再递归访问 右孩子。
3. 遍历时需要记录当前层节点之和。
4. 创建方法 inorder(node, level)，实现递归的中序遍历。该方法输入当前节点和当前节点层级，然后递归更新 level_sum[level]。
5. 返回数组 level_sum 的最大值。

```
 var maxLevelSum = function(root) {
    let arr=[]
    function dfs(node,depth){
        if(!node)return
        arr[depth]=arr[depth]||[]
        arr[depth].push(node.val)
        dfs(node.left,depth+1)
        dfs(node.right,depth+1)
    }
    dfs(root,0)
    let ans=[]
    for(let i=0;i<arr.length;i++){
        sum=arr[i].reduce((sum,cur)=>sum+cur)
        ans.push(sum)
    }
    let max=Math.max(...ans)
    return ans.indexOf(max)+1
};
```

10、572. 另一棵树的子树

1. 定义isSameTree()用来判断两个树是否相同
2. 首先判断s和t是否相同，相同返回true
3. 不相同就递归判断s的左右子树是否和t相同

```
 // 判断两个树是否相同
const isSameTree = (p, q) => {
    if (!p && !q) return true;
    if (p && q && p.val === q.val && isSameTree(q.left, p.left) &&
isSameTree(q.right, p.right)) {
        return true;
    }
    return false;
};

const isSubtree = (s, t) => {
    // 递归出口
    if (!s) return false;
    // 若两个树相同，肯定是true
    if (isSameTree(s, t)) return true;
    // 不相同，就递归判断s的左右子树
    return isSubtree(s.left, t) || isSubtree(s.right, t);
};
```

开课吧