1169. 查询无效交易（中等）

1. 题目描述

   如果出现下述两种情况，交易 **可能无效**：

   - 交易金额超过 ¥1000
   - 或者，它和另一个城市中同名的另一笔交易相隔不超过 60 分钟（包含 60 分钟整）

   每个交易字符串 `transactions[i]` 由一些用逗号分隔的值组成，这些值分别表示交易的名称，时间（以分钟计），金额以及城市。

   给你一份交易清单 `transactions`，返回可能无效的交易列表。你可以按任何顺序返回答案。

   示例 1：

   ```
   输入：transactions = ["alice,20,800,mtv","alice,50,100,beijing"]
   输出：["alice,20,800,mtv","alice,50,100,beijing"]
   解释：第一笔交易是无效的，因为第二笔交易和它间隔不超过 60 分钟、名称相同且发生在不同的城市。同样，第二笔交易也是无效的。
   ```

   示例 2：

   ```
   输入：transactions = ["alice,20,800,mtv","alice,50,1200,mtv"]
   输出：["alice,50,1200,mtv"]
   ```

   示例 3：

   ```
   输入：transactions = ["alice,20,800,mtv","bob,50,1200,mtv"]
   输出：["bob,50,1200,mtv"]
   ```

   提示：

   - `transactions.length <= 1000`
   - 每笔交易 `transactions[i]` 按 `"{name},{time},{amount},{city}"` 的格式进行记录
   - 每个交易名称 `{name}` 和城市 `{city}` 都由小写英文字母组成，长度在 `1` 到 `10` 之间
   - 每个交易时间 `{time}` 由一些数字组成，表示一个 `0` 到 `1000` 之间的整数
   - 每笔交易金额 `{amount}` 由一些数字组成，表示一个 `0` 到 `2000` 之间的整数

2. 简单实现

```cpp
typedef struct node{//存储一个交易
    string name;
    int time;
    int amount;
    string city;
};
class Solution {
public:
    vector<string> invalidTransactions(vector<string>& transactions) {
```

```
        vector<node*> v(transactions.size());//transactions[i]对应v[i]
        unordered_map<string, vector<int>> m;//<name, index>，各个name对应的所有
v/transactions中的索引
        unordered_set<int> s;//记录无效交易的索引，防止重复加入
        for(int i = 0; i < transactions.size(); i++){//构造v, m
            node* cur = new node;
            int l = 0, r = 0;
            while(transactions[i][r] != ',') r++;
            cur->name = transactions[i].substr(l, r-l);
            r++;
            l = r;
            while(transactions[i][r] != ',') r++;
            cur->time = stoi(transactions[i].substr(l, r-l));
            r++;
            l = r;
            while(transactions[i][r] != ',') r++;
            cur->amount = stoi(transactions[i].substr(l, r-l));
            r++;
            cur->city = transactions[i].substr(r, transactions[i].size() - r);
            v[i] = cur;//构造v
            m[cur->name].push_back(i);//构造m
            if(cur->amount > 1000)//这里加入，省的之后遍历再看
                s.insert(i);
        }

        for(auto it = m.begin(); it != m.end(); it++){//对所有同名交易
            for(int i = 0; i < it->second.size()-1; i++){//两两比较其城市和时间
                node* cur1 = v[it->second[i]];
                for(int j = i+1; j < it->second.size(); j++){
                    node* cur2 = v[it->second[j]];
                    if(abs(cur1->time-cur2->time) <= 60 && cur1->city!=cur2->city)
{//两者均无效
                        s.insert(it->second[i]);
                        s.insert(it->second[j]);
                    }
                }
            }
        }
        vector<string> ans;
        for(auto it = s.begin(); it != s.end(); it++)
            ans.push_back(transactions[*it]);
        return ans;
    }
};
```

1170. 比较字符串最小字母出现频次（简单）

1. 题目描述

   我们来定义一个函数 `f(s)`，其中传入参数 `s` 是一个非空字符串；该函数的功能是统计 `s` 中（按字典序比较）最小字母的出现频次。

   例如，若 `s = "dcce"`，那么 `f(s) = 2`，因为最小的字母是 `"c"`，它出现了 2 次。

现在，给你两个字符串数组待查表 `queries` 和词汇表 `words`，请你返回一个整数数组 `answer` 作为答案，其中每个 `answer[i]` 是满足 `f(queries[i])` < `f(W)` 的词的数目，`W` 是词汇表 `words` 中的词。

示例 1：

```
输入：queries = ["cbd"], words = ["zaaaz"]
输出：[1]
解释：查询 f("cbd") = 1，而 f("zaaaz") = 3 所以 f("cbd") < f("zaaaz")。
```

示例 2：

```
输入：queries = ["bbb","cc"], words = ["a","aa","aaa","aaaa"]
输出：[1,2]
解释：第一个查询 f("bbb") < f("aaaa")，第二个查询 f("aaa") 和 f("aaaa") 都 > f("cc")。
```

提示：

- `1 <= queries.length <= 2000`
- `1 <= words.length <= 2000`
- `1 <= queries[i].length, words[i].length <= 10`
- `queries[i][j]`，`words[i][j]` 都是小写英文字母

2. 简单实现

```
class Solution {
public:
    int getFre(string s){//获取f(s)
        sort(s.begin(), s.end());
        char c = s[0];
        int ans = 1;
        for(int i = 1; i < s.size(); i++){
            if(s[i] == c)
                ans++;
            else
                return ans;
        }
        return ans;
    }
    vector<int> numSmallerByFrequency(vector<string>& queries, vector<string>&
words) {
        vector<int> word_fre(words.size());
        for(int i = 0; i < words.size(); i++)
            word_fre[i] = getFre(words[i]);
        sort(word_fre.begin(), word_fre.end());
        vector<int> ans(queries.size());
        for(int i = 0; i < queries.size(); i++){
            int fre = getFre(queries[i]);
            int cnt = word_fre.end() - upper_bound(word_fre.begin(),
word_fre.end(), fre);
            ans[i] = cnt;
        }
        return ans;
    }
```

```
};
```

1171. 从链表中删去总和值为零的连续节点（中等）

1. 题目描述

   给你一个链表的头节点 `head`，请你编写代码，反复删去链表中由 **总和** 值为 `0` 的连续节点组成的序列，直到不存在这样的序列为止。

   删除完毕后，请你返回最终结果链表的头节点。

   你可以返回任何满足题目要求的答案。

   （注意，下面示例中的所有序列，都是对 `ListNode` 对象序列化的表示。）

   示例 1：

   ```
   输入：head = [1,2,-3,3,1]
   输出：[3,1]
   提示：答案 [1,2,1] 也是正确的。
   ```

   示例 2：

   ```
   输入：head = [1,2,3,-3,4]
   输出：[1,2,4]
   ```

   示例 3：

   ```
   输入：head = [1,2,3,-3,-2]
   输出：[1]
   ```

   提示：

   - 给你的链表中可能有 `1` 到 `1000` 个节点。
   - 对于链表中的每个节点，节点的值：`-1000 <= node.val <= 1000`.

2. 简单实现

   设sum[i]表示head[0...i]的和，则若 `sum[i] = sum[j], i<j`，则head[i+1...j]的和为0，需要删去，删去的方法为head[i]->next = head[j]->next即可，且删去这个序列不影响后面的元素的sum值

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        unordered_map<int, ListNode*> m;//记录sum值为int的ListNode
```

```
        ListNode* ans = new ListNode(0);//添加空的头节点简化操作
        ans->next = head;
        m[0] = ans;//为了处理从开头就存在0序列的情况
        int sum = 0;
        while(head){
            sum += head->val;
            if(m.count(sum) <= 0)
                m[sum] = head;
            else{//前面出现了sum值相同的节点，需要删除了
                //删除这之间的节点的sum值，以免后面出现相同的sum值时产生错误
                //这个过程中也应该释放删除的节点，这里省略了
                ListNode* cur = m[sum]->next;
                while(cur != head){
                    sum += cur->val;
                    m.erase(sum);
                    cur = cur->next;
                }
                sum += head->val;
                //删除0序列
                m[sum]->next = head->next;
            }
            head = head->next;
        }
        return ans->next;
    }
};
```

3. 最优解法

和为0序列间存在两种关系：

- 包含关系：例如 3，2，-2，3
- 平行关系：例如 3，-3，2，-2

```
class Solution {
public:
    ListNode* removeZeroSumSublists(ListNode* head) {
        ListNode* fake = new ListNode(-1);
        fake->next=head;
        ListNode* p = head;
        ListNode* a;
        ListNode* pre = fake;
        int sum=0;
        while(p!=nullptr) {
            a=p;
            sum=0;
            while(a!=nullptr) {
                sum+=a->val;
                if(sum==0) {//得到的一定是包含关系下最外围的和为0序列
                    pre->next=a->next;//删除
                    p=pre;//继续寻找平行关系的和为0序列
                }
                a=a->next;
            }
```

```
            pre = p;
            p=p->next;
        }
        return fake->next;
    }
};
```

1172. 餐盘栈（困难）

1. 题目描述

   我们把无限数量 ∞ 的栈排成一行，按从左到右的次序从 0 开始编号。每个栈的的最大容量 `capacity` 都相同。

   实现一个叫「餐盘」的类 `DinnerPlates`：

   - `DinnerPlates(int capacity)` - 给出栈的最大容量 `capacity`。
   - `void push(int val)` - 将给出的正整数 `val` 推入 **从左往右第一个** 没有满的栈。
   - `int pop()` - 返回 **从右往左第一个** 非空栈顶部的值，并将其从栈中删除；如果所有的栈都是空的，请返回 `-1`。
   - `int popAtStack(int index)` - 返回编号 `index` 的栈顶部的值，并将其从栈中删除；如果编号 `index` 的栈是空的，请返回 `-1`。

   示例：

```
输入：
["DinnerPlates","push","push","push","push","push","popAtStack","push","push","popAtStack","popAtStack","pop","pop","pop","pop","pop"]
[[2],[1],[2],[3],[4],[5],[0],[20],[21],[0],[2],[],[],[],[],[]]
输出：
[null,null,null,null,null,null,2,null,null,20,21,5,4,3,1,-1]

解释：
DinnerPlates D = DinnerPlates(2);  // 初始化，栈最大容量 capacity = 2
D.push(1);
D.push(2);
D.push(3);
D.push(4);
D.push(5);         // 栈的现状为：    2  4
                                    1  3  5
                                    ﹈ ﹈ ﹈
D.popAtStack(0);   // 返回 2。栈的现状为：      4
                                          1  3  5
                                          ﹈ ﹈ ﹈
D.push(20);        // 栈的现状为：  20  4
                                   1  3  5
                                   ﹈ ﹈ ﹈
D.push(21);        // 栈的现状为：  20  4 21
                                   1  3  5
                                   ﹈ ﹈ ﹈
D.popAtStack(0);   // 返回 20。栈的现状为：       4 21
                                            1  3  5
```

```
                                                  ┌─┐ ┌─┐ ┌─┐
D.popAtStack(2);    // 返回 21。栈的现状为:          4
                                                  1 │ 3 │ 5
                                                  └─┘ └─┘ └─┘

D.pop()             // 返回 5。栈的现状为:           4
                                                  1 │ 3 │
                                                  └─┘ └─┘

D.pop()             // 返回 4。栈的现状为:       1 │ 3 │
                                                  └─┘ └─┘

D.pop()             // 返回 3。栈的现状为:       1 │
                                                  └─┘

D.pop()             // 返回 1。现在没有栈。
D.pop()             // 返回 -1。仍然没有栈。
```

提示：

- `1 <= capacity <= 20000`
- `1 <= val <= 20000`
- `0 <= index <= 100000`
- 最多会对 `push` ， `pop` ，和 `popAtStack` 进行 `200000` 次调用。

2. 简单实现

```
class DinnerPlates {
public:
    vector<stack<int>> stacks;//保存所有的栈
    int capacity;
    //维护两个idx防止stacks中不断的删栈加栈
    int l_idx;//维护push时即将进入的栈idx
    int r_idx;//维护pop时即将pop的栈idx
    DinnerPlates(int capacity) {
        this->capacity = capacity;
        l_idx = 0;
        r_idx = -1;
    }

    void push(int val) {
        if(l_idx == stacks.size()){//当前stacks中的栈个数不够用了
            stack<int> cur;
            cur.push(val);
            stacks.push_back(cur);
            r_idx = l_idx;
        }
        else
            stacks[l_idx].push(val);
        if(stacks[l_idx].size() == capacity){//栈满
            l_idx++;
            while(l_idx < stacks.size() && stacks[l_idx].size() == capacity)//从左到
右找到第一个未满的栈
                l_idx++;
        }
    }

    int pop() {
```

```
        if(r_idx == -1)
            return -1;
        int ans = stacks[r_idx].top();
        stacks[r_idx].pop();
        while(r_idx >= 0 && stacks[r_idx].empty())//从右到左找到第一个非空栈
            r_idx--;
        return ans;
    }

    int popAtStack(int index) {
        if(index >= stacks.size() || stacks[index].empty())
            return -1;
        int ans = stacks[index].top();
        stacks[index].pop();
        if(l_idx > index)//pop的栈位于l_idx左侧
            l_idx = index;
        while(r_idx >= 0 && stacks[r_idx].empty())
            r_idx--;
        return ans;
    }
};
```