1281.整数的各位积和之差（简单）

太简单，略

1282.用户分组（中等）

1. 题目描述

   有 n 位用户参加活动，他们的 ID 从 0 到 n - 1，每位用户都 恰好 属于某一用户组。给你一个长度为 n 的数组 groupSizes，其中包含每位用户所处的用户组的大小，请你返回用户分组情况（存在的用户组以及每个组中用户的 ID）。你可以任何顺序返回解决方案，ID 的顺序也不受限制。此外，题目给出的数据保证至少存在一种解决方案。

   ```
   示例 1:
   输入: groupSizes = [3,3,3,3,3,1,3]
   输出: [[5],[0,1,2],[3,4,6]]
   解释:
   其他可能的解决方案有 [[2,1,6],[5],[0,4,3]] 和 [[5],[0,6,2],[4,3,1]]。

   示例 2:
   输入: groupSizes = [2,1,3,3,3,2]
   输出: [[1],[0,5],[2,3,4]]
   ```

提示:

- groupSizes.length == n
- 1 <= n <= 500
- 1 <= groupSizes[i] <= n

2. 比赛中实现

   因为保证了一定有解，所以用map保存属于长度为n的数组的id，每够n个就push到答案中即可

```
class Solution {
public:
    vector<vector<int>> groupThePeople(vector<int>& groupSizes) {
        vector<vector<int>> ans;
        unordered_map<int, vector<int>> m;//<size, id>
        for(int i = 0; i < groupSizes.size(); i++){
            if(m.count(groupSizes[i]) <= 0)
                if(groupSizes[i] > 1)
                    m[groupSizes[i]] = {i};
                else
                    ans.push_back({i});
            else{
                m[groupSizes[i]].push_back(i);
                if(m[groupSizes[i]].size() == groupSizes[i]){//够个数了
                    ans.push_back(m[groupSizes[i]]);
                    m.erase(groupSizes[i]);
                }
```

```
                }
            }
            return ans;
        }
};
```

3. 最优解法

用0标记被放入ans的id，每次找到前n个属于长度为n的数组的id放入ans，这样省去map占用的空间和操作时间，但增加了一些线性查找的时间

```
class Solution {
public:
    vector<vector<int>> groupThePeople(vector<int>& groupSizes) {
        int n=groupSizes.size();
        vector<vector<int >> ans;
        if(n!=0) {
            for(int i=0;i<n;i++) {
                int num=groupSizes[i];
                if(num==0) continue;//已经被放入ans中
                else if(num==1) {//直接放入
                    vector<int> tmp {i};
                    ans.push_back(tmp);
                }
                else{
                    vector<int > tmp(num);
                    tmp[--num] = i;
                    for(int j=i+1; j<n && num>0; j++) {//在剩下的id中找到同组的放进ans
                        if(groupSizes[i]==groupSizes[j]) {
                            groupSizes[j]=0;//已经放入ans，标记为0
                            tmp[--num]=j;
                        }
                    }
                   ans.push_back(tmp);
                }
            }
        }
        return ans;
    }
};
```

1283.使结果不超过阈值的最小除数（中等）

1. 题目描述

给你一个整数数组 nums 和一个正整数 threshold ，你需要选择一个正整数作为除数，然后将数组里每个数都除以它，并对除法结果求和。请你找出能够使上述结果小于等于阈值 threshold 的除数中 最小 的那个。

每个数除以除数后都向上取整，比方说 7/3 = 3 ， 10/2 = 5 。

题目保证一定有解。

```
示例 1:
输入: nums = [1,2,5,9], threshold = 6
输出: 5
解释: 如果除数为 1 ，我们可以得到和为 17 (1+2+5+9)。
如果除数为 4 ，我们可以得到和为 7 (1+1+2+3) 。如果除数为 5 ，和为 5 (1+1+1+2)。

示例 2:
输入: nums = [2,3,5,7,11], threshold = 11
输出: 3

示例 3:
输入: nums = [19], threshold = 5
输出: 4
```

提示:

- 1 <= nums.length <= 5 * 10^4
- 1 <= nums[i] <= 10^6
- nums.length <= threshold <= 10^6

2. 比赛中实现

二分法

```cpp
class Solution {
public:
    int judge(vector<int>& nums, int threshold, int divider){//判断当前divider的结果是否大于阈值
        int sum = 0;
        for(int i = 0; i < nums.size(); i++){
            sum += (nums[i]+divider-1) / divider;//向上取整的除法
            if(sum > threshold) return 1;
        }
        return -1;
    }
    int smallestDivisor(vector<int>& nums, int threshold) {
        if(nums.size() == 1)
            return (nums[0]+threshold-1) / threshold;//向上取整的除法
        int l = 1, r = 1e6;
        while(l < r){
            int mid = l + (r - l) / 2;
            int re = judge(nums, threshold, mid);
            if(re > 0) l = mid + 1;
            else r = mid;
        }
        return l;
    }
};
```

1284.转化为全零矩阵的最少反转次数（困难）

1. 题目描述

给你一个 m x n 的二进制矩阵 mat。每一步，你可以选择一个单元格并将它反转（反转表示 0 变 1 ，1 变 0 ）。如果存在和它相邻的单元格，那么这些相邻的单元格也会被反转。（注：相邻的两个单元格共享同一条边。）

请你返回将矩阵 mat 转化为全零矩阵的最少反转次数，如果无法转化为全零矩阵，请返回 -1 。

二进制矩阵的每一个格子要么是 0 要么是 1 。

全零矩阵是所有格子都为 0 的矩阵。

```
示例 1:
输入: mat = [[0,0],[0,1]]
输出: 3
解释: 一个可能的解是反转 (1, 0)，然后 (0, 1) ，最后是 (1, 1) 。

示例 2:
输入: mat = [[0]]
输出: 0
解释: 给出的矩阵是全零矩阵，所以你不需要改变它。

示例 3:
输入: mat = [[1,1,1],[1,0,1],[0,0,0]]
输出: 6

示例 4:
输入: mat = [[1,0,0],[1,0,0]]
输出: -1
解释: 该矩阵无法转变成全零矩阵
```

提示:

- m == mat.length
- n == mat[0].length
- 1 <= m <= 3
- 1 <= n <= 3
- mat_{ij} 是 0 或 1 。

2. 比赛中实现

居然想出来还顺利写出来了，我真是个天才哈哈哈哈！！！心路历程：

- 啊呀最多才三行三列诶，可以枚举不? —— $2^9 = 512$枚举你妹啊
- 动态规划？从小块找规律推广到大块？——没找到
- 从给定的矩阵出发，进行广度优先搜索找到全零矩阵
  - 问题一：如何标记已经搜索过的矩阵？——集合，但由于vector无法哈希化，将其序列化为字符串形式存储
  - 问题二：何时终止?
    - 因为存在无解的情况，必须找到终止并返回-1的条件但我发现我
    - 可以想象，并不是出现重复矩阵了就代表无解了，而应该是经过一轮广度优先遍历后，集合的大小没有出现变化，才说明无解了（相当于这一步所有的可能情况都在之前出现过并且否定过，不可能再出现新的情况了）
  - 此外在队列中也使用序列化，即能节省存储空间，还能节约反复创建矩阵的时间

```
class Solution {
```

```
public:
    int m;
    int n;
    string mat2str(const vector<vector<int>>& mat){//矩阵序列化为字符串
        string ans = "";
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                ans += mat[i][j] + '0';
        return ans;
    }
    vector<vector<int>> str2mat(const string& s){//字符串反序列化为矩阵
        vector<vector<int>> ans = vector<vector<int>>(m, vector<int>(n));
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                ans[i][j] = s[i*n + j] - '0';
        return ans;
    }
    void reverse(vector<vector<int>>& mat, int x, int y){//反转mat(x,y)
        mat[x][y] = (mat[x][y]+1) % 2;
        if(x-1 >= 0) mat[x-1][y] = (mat[x-1][y]+1) % 2;
        if(x+1 < m) mat[x+1][y] = (mat[x+1][y]+1) % 2;
        if(y-1 >= 0) mat[x][y-1] = (mat[x][y-1]+1) % 2;
        if(y+1 < n) mat[x][y+1] = (mat[x][y+1]+1) % 2;
    }
    int minFlips(vector<vector<int>>& mat) {
        m = mat.size();
        n = mat[0].size();

        vector<vector<int>> cur = vector<vector<int>>(m, vector<int>(n, 0));//全零矩
阵
        string aim_str = mat2str(cur);//目标全零矩阵的序列化
        string mat_str = mat2str(mat);//输入矩阵的序列化
        if(mat_str == aim_str) return 0;//当前就是全0矩阵

        queue<string> q;//广度优先遍历用队列
        q.push(mat_str);
        unordered_set<string> s;//记录已访问矩阵
        s.insert(mat_str);
        int cnt = 0;//当前反转次数
        while(1){
            int q_size = q.size();
            int s_size = s.size();
            cnt++;
            for(int i = 0; i < q_size; i++){
                string temp = q.front();
                cur = str2mat(temp);
                q.pop();
                //遍历对当前矩阵每个位置依次做反转的情况
                for(int x = 0; x < m; x++)
                    for(int y = 0; y < n; y++){
                        reverse(cur, x, y);//反转cur(x,y)
                        string cur_str = mat2str(cur);
                        if(cur_str == aim_str) return cnt;//找到
```

```
                        else{
                            q.push(mat2str(cur));
                            s.insert(cur_str);
                        }
                        reverse(cur, x, y);//反转回原始状态
                    }
                }
                if(s.size() == s_size) return -1;//集合不再增长，说明无解了
            }
            return -1;
        }
};
```

3. 自我改进

为何不把reverse操作也放在字符串内呢？这样就省去来回转换了

```
class Solution {
public:
    int m;
    int n;
    string mat2str(const vector<vector<int>>& mat){
        string ans = "";
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                ans += mat[i][j] + '0';
        return ans;
    }
    void reverse(string& s, int x, int y){
        s[x*n + y] = s[x*n + y] == '0' ? '1' : '0';
        if(x-1 >= 0) s[(x-1)*n + y] = s[(x-1)*n + y] == '0' ? '1' : '0';
        if(x+1 < m) s[(x+1)*n + y] = s[(x+1)*n + y] == '0' ? '1' : '0';
        if(y-1 >= 0) s[x*n + y-1] = s[x*n + y-1] == '0' ? '1' : '0';
        if(y+1 < n) s[x*n + y+1] = s[x*n + y+1] == '0' ? '1' : '0';
    }
    int minFlips(vector<vector<int>>& mat) {
        m = mat.size();
        n = mat[0].size();

        vector<vector<int>> temp = vector<vector<int>>(m, vector<int>(n, 0));//全零
矩阵
        string aim_str = mat2str(temp);//目标全零矩阵的序列化
        string mat_str = mat2str(mat);//输入矩阵的序列化
        if(mat_str == aim_str) return 0;//当前就是全0矩阵

        queue<string> q;//广度优先遍历用队列
        q.push(mat_str);
        unordered_set<string> s;//记录已访问矩阵
        s.insert(mat_str);
        int cnt = 0;//当前反转次数
        while(1){
            int q_size = q.size();
            int s_size = s.size();
```

```
            cnt++;
            for(int i = 0; i < q_size; i++){
                string cur = q.front();
                q.pop();
                //遍历对当前矩阵每个位置依次做反转的情况
                for(int x = 0; x < m; x++)
                    for(int y = 0; y < n; y++){
                        reverse(cur, x, y);//反转cur(x,y)
                        if(cur == aim_str) return cnt;//找到
                        else{
                            q.push(cur);
                            s.insert(cur);
                        }
                        reverse(cur, x, y);//反转回原始状态
                    }
            }
            if(s.size() == s_size) return -1;//集合不再增长，说明无解了
        }
        return -1;
    }
};
```

4. 最优解法——深度优先遍历

假设二维矩阵 mat 经过 k 次翻转操作 T1, T2, ..., Tk 后变成全零矩阵，且 k 是最少的翻转次数，那么我们可以发现如下两个性质：

- 顺序无关性： 将这 k 次翻转操作任意打乱顺序，再从二维矩阵 mat 开始依次进行这些翻转操作，最后仍然会得到全零矩阵。证明如下：对于 mat 中的任意位置 (x, y)，假设 T1, T2, ..., Tk 中有 c 次翻转操作会将其进行翻转，那么无论操作的顺序怎么被打乱，位置 (x, y) 被翻转的次数总是为 c 次，即位置 (x, y) 的最终状态不会受到操作顺序的影响。
- 翻转唯一性： 这 k 次翻转操作中，不会有两次翻转操作选择了相同的位置。证明方法类似：对于 mat 中的任意位置 (x, y)，假设 T1, T2, ..., Tk 中有 c 次翻转操作会将其进行翻转，如果有两次翻转操作选择了相同的位置，那么将它们一起移除后，位置 (x, y) 要么被翻转 c 次，要么被翻转 c - 2 次。而减少一个位置 2 次翻转次数，对该位置不会有任何影响，即位置 (x, y) 的最终状态不会受到移除这两次翻转操作的影响。这样我们只使用 k - 2 次翻转操作，就可以得到全零矩阵，与 k 是最少的翻转次数矛盾。

根据上面两个结论，我们可以发现：对于二维矩阵 mat，如果我们希望它通过最少的翻转操作得到全零矩阵，那么 mat 中每个位置至多被翻转一次，并且翻转的顺序不会影响最终的结果。这样以来，翻转的方法一共只有 2^{MN}种，即对于 mat 中的每个位置，可以选择翻转或不翻转 2 种情况，位置的数量为 MN。

因此我们可以使用深度优先搜索来枚举所有的翻转方法。我们按照行优先的顺序搜索二维矩阵 mat 中的每个位置，对于当前位置，我们可以选择翻转或不翻转。当搜索完 mat 的所有位置时，如果 mat 变成了全零矩阵，那么我们就找到了一种满足要求的翻转方法。在所有满足要求的方法中，操作次数最少的即为答案。

此外，该解法把矩阵序列化为int数字了

```
class Solution {
private:
    static constexpr int dirs[5][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}, {0, 0}};
    int ans;
public:
    Solution(): ans(1e9) {
    }
```

```cpp
    void convert(vector<vector<int>>& mat, int m, int n, int i, int j) {
        for (int k = 0; k < 5; ++k) {
            int i0 = i + dirs[k][0], j0 = j + dirs[k][1];
            if (i0 >= 0 && i0 < m && j0 >= 0 && j0 < n) {
                mat[i0][j0] ^= 1;
            }
        }
    }

    void dfs(vector<vector<int>>& mat, int m, int n, int pos, int flip_cnt) {
        if (pos == m * n) {
            bool flag = true;
            for (int i = 0; i < m; ++i) {
                for (int j = 0; j < n; ++j) {
                    if (mat[i][j] != 0) {
                        flag = false;
                    }
                }
            }
            if (flag) {
                ans = min(ans, flip_cnt);
            }
            return;
        }

        int x = pos / n, y = pos % n;
        // not flip
        dfs(mat, m, n, pos + 1, flip_cnt);
        // flip
        convert(mat, m, n, x, y);
        dfs(mat, m, n, pos + 1, flip_cnt + 1);
        convert(mat, m, n, x, y);
    }
    int minFlips(vector<vector<int>>& mat) {
        int m = mat.size(), n = mat[0].size();
        dfs(mat, m, n, 0, 0);
        return (ans != 1e9 ? ans : -1);
    }
};
```