Day 37 / 100 :

Topic - Matrix

1 Problem statement: Last Stone weight II (Medium)

You are given an array of integers stones where stones[i] is the weight of the ith stone.
We are playing a game with the stones. On each turn, we choose any two stones and smash them together. Suppose the stones have weights x and y with x <= y. The result of this smash is:

- If x == y, both stones are destroyed, and
- If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is at most one stone left.
Return the smallest possible weight of the left stone. If there are no stones left, return 0.

Example 1:
Input: stones = [2,7,4,1,8,1]
Output: 1
Explanation:
We can combine 2 and 4 to get 2, so the array converts to [2,7,1,8,1] then,
we can combine 7 and 8 to get 1, so the array converts to [2,1,1,1] then,
we can combine 2 and 1 to get 1, so the array converts to [1,1,1] then,
we can combine 1 and 1 to get 0, so the array converts to [1], then that's the optimal value.

Example 2:
Input: stones = [31,26,33,21,40]
Output: 5

Solutions :

Approach 1 - 2D Array + Maths

As difference of sum between two subset should be minimum, let SUM = total sum of array, sum1 and sum2 be the sum of the two subsets:

abs(sum1-sum2) should be minimum.

sum1 = SUM -sum2, so we can say (SUM-sum2-sum2) should be min.

for SUM - 2*sum2 to be minimum,* 2sum2 should be maximum.

2sum2 can be less than or equal to SUM,

so 2sum2<=SUM.
sum2<=SUM/2.

So we can apply DP appraoch of subset sum problem. Let t[][] be the matrix of DP, if sum2=SUM/2, t[n][sum2] will give the sum of one subset, if sum2<SUM/2, we can traverse in the t[n][j] row backwards, to check the possibility of sum and can get the sum of one subset.

sum = SUM/2,

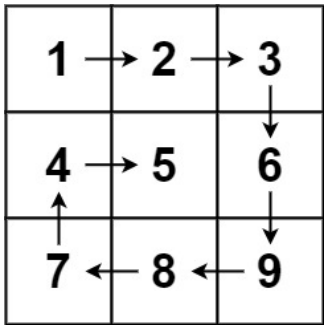
```
int n = nums.size(),SUM=0;
 for(int i=0;i<n;i++){
     SUM+=nums[i];
 }int sum = SUM/2;
 bool t[n+1][abs(sum+1)];
 for(int i=0;i<=n;i++) t[i][0]=true;
 for(int j=1;j<=sum;j++) t[0][j]=false;

 for(int i=1;i<=n;i++){
     for(int j=1;j<=sum;j++){
         if(nums[i-1]<=j) t[i][j] = t[i-1][j-nums[i-1]] || t[i-1][j];
         else t[i][j] = t[i-1][j];
     }
 }
 for(int i=sum;i>=0;i--){
     if(t[n][i]==true) return SUM -2*i;
 }
 return 0;
```

2 Problem statement: Spiral Matrix (Medium)

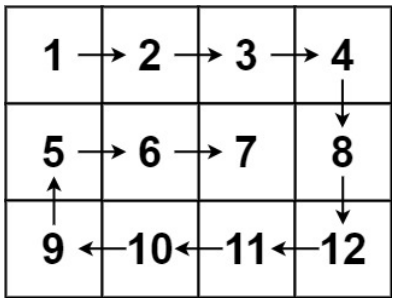
Given an m x n matrix, return all elements of the matrix in spiral order.

Example 1:



Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]

Example 2:



Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]

Solutions :

Approach 1 - 2D Array + Maths

When traversing the matrix in the spiral order, at any time we follow one out of the following four directions: RIGHT DOWN LEFT UP. Suppose we are working on a 5 x 3 matrix as such:

0 1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

Imagine a cursor starts off at (0, -1), i.e. the position at '0', then we can achieve the spiral order by doing the following:

1. Go right 5 times

2. Go down 2 times
3. Go left 4 times
4. Go up 1 times.
5. Go right 3 times
6. Go down 0 times -> quit

Notice that the directions we choose always follow the order 'right->down->left->up', and for horizontal movements, the number of shifts follows:{5, 4, 3}, and vertical movements follows {2, 1, 0}.

Thus, we can make use of a direction matrix that records the offset for all directions, then an array of two elements that stores the number of shifts for horizontal and vertical movements, respectively. This way, we really just need one for loop instead of four.

Another good thing about this implementation is that: If later we decided to do spiral traversal on a different direction (e.g. Counterclockwise), then we only need to change the Direction matrix; the main loop does not need to be touched.

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
    vector<vector<int> > dirs{{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    vector<int> res;
    int nr = matrix.size();     if (nr == 0) return res;
    int nc = matrix[0].size();  if (nc == 0) return res;

    vector<int> nSteps{nc, nr-1};

    int iDir = 0;   // index of direction.
    int ir = 0, ic = -1;    // initial position
    while (nSteps[iDir%2]) {
        for (int i = 0; i < nSteps[iDir%2]; ++i) {
            ir += dirs[iDir][0]; ic += dirs[iDir][1];
            res.push_back(matrix[ir][ic]);
        }
        nSteps[iDir%2]--;
        iDir = (iDir + 1) % 4;
    }
    return res;
}
```