1275.找出井字棋的获胜者（简单）

1. 题目描述

   A 和 *B* 在一个 *3* x *3* 的网格上玩井字棋。

   井字棋游戏的规则如下：

   - 玩家轮流将棋子放在空方格 (" ") 上。
   - 第一个玩家 A 总是用 "X" 作为棋子，而第二个玩家 B 总是用 "O" 作为棋子。
   - "X" 和 "O" 只能放在空方格中，而不能放在已经被占用的方格上。
   - 只要有 3 个相同的（非空）棋子排成一条直线（行、列、对角线）时，游戏结束。
   - 如果所有方块都放满棋子（不为空），游戏也会结束。
   - 游戏结束后，棋子无法再进行任何移动。

   给你一个数组 `moves`，其中每个元素是大小为 `2` 的另一个数组（元素分别对应网格的行和列），它按照 *A* 和 *B* 的行动顺序（先 *A* 后 *B*）记录了两人各自的棋子位置。如果游戏存在获胜者（*A* 或 *B*），就返回该游戏的获胜者；如果游戏以平局结束，则返回 "Draw"；如果仍会有行动（游戏未结束），则返回 "Pending"。

   你可以假设 `moves` 都 **有效**（遵循井字棋规则），网格最初是空的，*A* 将先行动。

   示例 1:

   ```
   输入：moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]
   输出："A"
   ```

   示例 2:

   ```
   输入：moves = [[0,0],[1,1],[0,1],[0,2],[1,0],[2,0]]
   输出："B"
   ```

   示例 3:

   ```
   输入：moves = [[0,0],[1,1],[2,0],[1,0],[1,2],[2,1],[0,1],[0,2],[2,2]]
   输出："Draw"
   输出：由于没有办法再行动，游戏以平局结束。
   "XXO"
   "OOX"
   "XOX"
   ```

   示例 4:

   ```
   输入：moves = [[0,0],[1,1]]
   输出："Pending"
   解释：游戏还没有结束。
   "X  "
   " O "
   "   "
   ```

提示：

- `1 <= moves.length <= 9`
- `moves[i].length == 2`
- `0 <= moves[i][j] <= 2`
- `moves` 里没有重复的元素。
- `moves` 遵循井字棋的规则。

2. 比赛时实现

   数据量少时暴力枚举即可，数据量大时可以用位运算

```
class Solution {
public:
    string tictactoe(vector<vector<int>>& moves) {
        vector<vector<string>> v = vector<vector<string>>(3, vector<string>(3, " "));
        for(int i = 0; i < moves.size(); i++)
            v[moves[i][0]][moves[i][1]] = i%2 == 0? "A" : "B";

        for(int i = 0; i < 3; i++)
            if(v[i][0] != " " && v[i][0] == v[i][1] && v[i][1] == v[i][2]) return v[i][0];
        for(int j = 0; j < 3; j++)
            if(v[0][j] != " " && v[0][j] == v[1][j] && v[1][j] == v[2][j]) return v[0][j];
        if(v[0][0] != " " && v[0][0] == v[1][1] && v[1][1] == v[2][2]) return v[0][0];
        if(v[0][2] != " " && v[0][2] == v[1][1] && v[1][1] == v[2][0]) return v[0][2];
        if(moves.size() == 9) return "Draw";
        else return "Pending";
    }
};
```

1276.不浪费原料的汉堡制作方案（中等）

1. 题目描述

   圣诞活动预热开始啦，汉堡店推出了全新的汉堡套餐。为了避免浪费原料，请你帮他们制定合适的制作计划。

   给你两个整数 `tomatoSlices` 和 `cheeseSlices`，分别表示番茄片和奶酪片的数目。不同汉堡的原料搭配如下：

   - **巨无霸汉堡：** 4 片番茄和 1 片奶酪
   - **小皇堡：** 2 片番茄和 1 片奶酪

   请你以 `[total_jumbo, total_small]`（[巨无霸汉堡总数，小皇堡总数]）的格式返回恰当的制作方案，使得剩下的番茄片 `tomatoSlices` 和奶酪片 `cheeseSlices` 的数量都是 `0`。

   如果无法使剩下的番茄片 `tomatoSlices` 和奶酪片 `cheeseSlices` 的数量为 `0`，就请返回 `[]`。

   示例 1：

```
输入：tomatoSlices = 16, cheeseSlices = 7
输出：[1,6]
解释：制作 1 个巨无霸汉堡和 6 个小皇堡需要 4*1 + 2*6 = 16 片番茄和 1 + 6 = 7 片奶酪。不会剩下原料。
```

示例 2：

```
输入：tomatoSlices = 17, cheeseSlices = 4
输出：[]
解释：只制作小皇堡和巨无霸汉堡无法用光全部原料。
```

示例 3：

```
输入：tomatoSlices = 4, cheeseSlices = 17
输出：[]
解释：制作 1 个巨无霸汉堡会剩下 16 片奶酪，制作 2 个小皇堡会剩下 15 片奶酪。
```

提示：

- `0 <= tomatoSlices <= 10^7`
- `0 <= cheeseSlices <= 10^7`

2. 比赛时实现

   就是求方程组

```
x+y = b;
4x+2y =a
```

   有没有x >= 0且 y >= 0的整数解

```cpp
class Solution {
public:
    vector<int> numOfBurgers(int tomatoSlices, int cheeseSlices) {
        int x = tomatoSlices - 2*cheeseSlices;
        if(x >= 0 && x%2 == 0){
            x /= 2;
            if(x <= cheeseSlices)
                return {x, cheeseSlices-x};
        }
        return {};
    }
};
```

1277.统计全为1的正方形子矩阵（中等）

1. 题目描述

   给你一个 `m * n` 的矩阵，矩阵中的元素不是 `0` 就是 `1`，请你统计并返回其中完全由 `1` 组成的 **正方形** 子矩阵的个数。

示例 1：

```
输入：matrix =
[
  [0,1,1,1],
  [1,1,1,1],
  [0,1,1,1]
]
输出：15
解释：
边长为 1 的正方形有 10 个。
边长为 2 的正方形有 4 个。
边长为 3 的正方形有 1 个。
正方形的总数 = 10 + 4 + 1 = 15.
```

示例 2：

```
输入：matrix =
[
  [1,0,1],
  [1,1,0],
  [1,1,0]
]
输出：7
解释：
边长为 1 的正方形有 6 个。
边长为 2 的正方形有 1 个。
正方形的总数 = 6 + 1 = 7.
```

提示：

- `1 <= arr.length <= 300`
- `1 <= arr[0].length <= 300`
- `0 <= arr[i][j] <= 1`

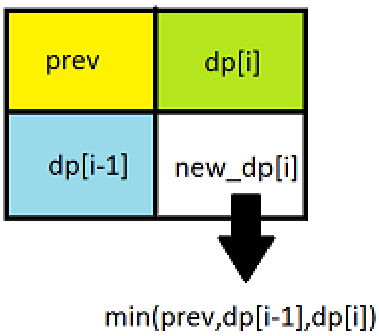
2. 比赛时实现

发现暴力搜索的复杂度足够，于是直接暴力了

```
class Solution {
public:
    int countSquares(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        int max_square_len = min(m, n);//子矩阵最大的可能的边长
        int cnt = 0;
        for(int len = 1; len <= max_square_len; len++)//从边长为1找起
            for(int i = 0; i <= m-len; i++)//子矩阵左上角行坐标
                for(int j = 0; j <= n-len; j++){//子矩阵左上角列坐标
                    bool flag = true;//是否全为1
                    //遍历当前对应的子矩阵
                    for(int x = i; flag && x < i+len; x++)
                        for(int y  = j; y < j+len; y++){
```

```
                            if(matrix[x][y] == 0){
                                flag = false;
                                break;
                            }
                        }
                    if(flag) cnt++;
                }
            return cnt;
        }
    };
```

3. 最优解法——动态规划



参考的是221最大正方形问题（见下一补充题），建议做完那题看完官方dp优化，非常巧妙的利用了空间。

本题相对221题只是多了一步累加所有dp_{ij}而已，因为实际上dp_{ij}也可以看作是以位置(i, j)为右下角的正方形的数目

```
class Solution {
public:
    int countSquares(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m==0) return 0;
        int n = matrix[0].size();
        int re = 0;
        int pre = 0;
        vector<int> dp(n+1,0);
        for(int i =1;i<=m;i++){
            for(int j =1;j<=n;j++){
                int temp = dp[j];
                if(matrix[i-1][j-1]==1){
                    dp[j] = min(min(dp[j],dp[j-1]),pre) + 1;
                    re += dp[j];
                }else dp[j] = 0;
                pre = temp;
            }
        }
        return re;
```

```
        }
    };
```

补充——221.最大正方形（中等）

1. 题目描述

   在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

   ```
   示例:

   输入:
   1 0 1 0 0
   1 0 1 1 1
   1 1 1 1 1
   1 0 0 1 0

   输出: 4
   ```

2. 官方题解

   dp_{ij}可以理解为以每个值为1的位置为左下角的最大全1矩形的边长

方法二：动态规划

我们用一个例子来解释这个方法：

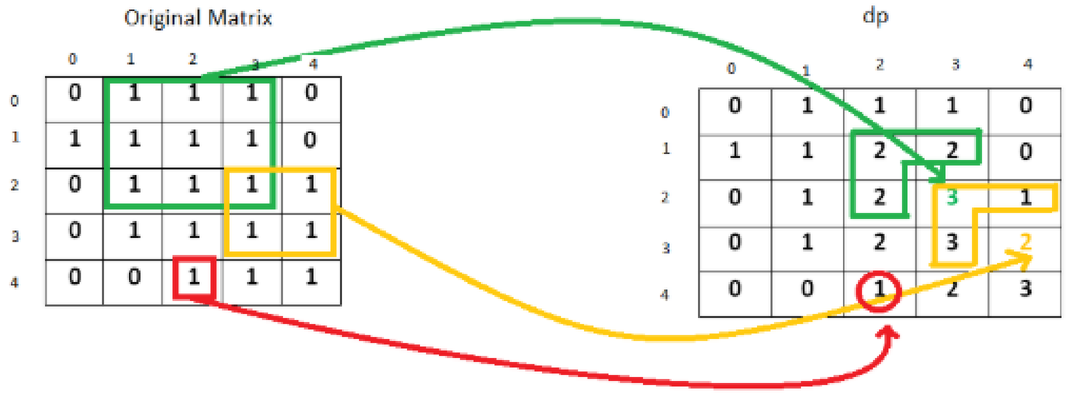
```
0 1 1 1 0
1 1 1 1 1
0 1 1 1 1
0 1 1 1 1
0 0 1 1 1
```

1. 我们用 0 初始化另一个矩阵 `dp`，维数和原始矩阵维数相同；
2. `dp(i,j)` 表示的是由 1 组成的最大正方形的边长；
3. 从 $(0,0)$ 开始，对原始矩阵中的每一个 1，我们将当前元素的值更新为

$$\mathrm{dp}(i,\ j) = \min\left(\mathrm{dp}(i-1,\ j),\ \mathrm{dp}(i-1,\ j-1),\ \mathrm{dp}(i,\ j-1)\right) + 1$$

4. 我们还用一个变量记录当前出现的最大边长，这样遍历一次，找到最大的正方形边长 $maxsqlen$，那么结果就是 $maxsqlen^2$。

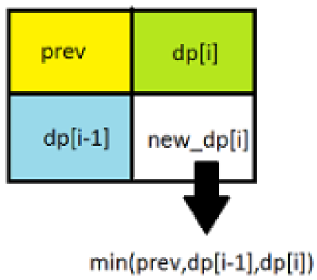
可以通过下面的图来理解该工作原理：



方法三：动态规划优化

在前面的动态规划解法中，计算 i^{th} 行（row）的 dp 方法中，我们只使用了上一个元素和第 $(i-1)^{th}$ 行，因此我们不需要二维 dp 矩阵，因为一维 dp 足以满足此要求。

我们扫描一行原始矩阵元素时，我们根据公式：$dp[j] = min(dp[j-1], dp[j], prev)$ 更新数组 dp，其中 prev 指的是 $dp[j-1]$，对于每一行，我们重复相同过程并在 dp 矩阵中更新元素。



TIPS：感觉图画错了，黄色的prev应该和蓝色的dp[i-1]换一下吧?

1278.分割回文串 III（困难）

1. 题目描述

   给你一个由小写字母组成的字符串 `s`，和一个整数 `k`。

   请你按下面的要求分割字符串：

   - 首先，你可以将 `s` 中的部分字符修改为其他的小写英文字母。
   - 接着，你需要把 `s` 分割成 `k` 个非空且不相交的子串，并且每个子串都是回文串。

   请返回以这种方式分割字符串所需修改的最少字符数。

   示例 1：

   ```
   输入：s = "abc", k = 2
   输出：1
   解释：你可以把字符串分割成 "ab" 和 "c"，并修改 "ab" 中的 1 个字符，将它变成回文串。
   ```

   示例 2：

   ```
   输入：s = "aabbc", k = 3
   输出：0
   解释：你可以把字符串分割成 "aa"、"bb" 和 "c"，它们都是回文串。
   ```

   示例 3：

   ```
   输入：s = "leetcode", k = 8
   输出：0
   ```

   提示：

   - `1 <= k <= s.length <= 100`
   - `s` 中只含有小写英文字母。

2. 比赛时实现

   用的bfs + dfs，疯狂超时

   ```
   class Solution {
   public:
       vector<vector<int>> locs;//每个字母出现的位置
       bool judge(string& s, int l, int r){//s[l...r]是否回文
           while(l < r){
               if(s[l] != s[r]) return false;
               l++;
               r--;
           }
           return true;
       }
       bool judgek(string& s, int l, int r, int k){//能否把s[l...r]分割成k个回文串
           if(k == r-l+1) return true;
           if(k == 1) return judge(s, l, r);

           vector<int> loc = locs[s[l]-'a'];

           for(int i = 0; i < loc.size(); i++)
   ```

```
            if(loc[i] >= l){
                if(judge(s, l, loc[i]) && judgek(s, loc[i]+1, r, k-1))
                    return true;
            }
        return false;
    }
    int palindromePartition(string s, int k) {
        if(k == s.size()) return 0;
        unordered_set<char> has;
        locs = vector<vector<int>>(26);
        for(int i = 0; i < s.size(); i++){
            has.insert(s[i]);
            locs[s[i]-'a'].push_back(i);
        }
        if(judgek(s, 0, s.size()-1, k)) return 0;
        int cnt = 0;
        queue<string> q;
        q.push(s);
        unordered_set<string> visited;
        visited.insert(s);
        while(!q.empty()){
            cnt++;
            int size = q.size();
            for(int i = 0; i < size; i++){
                string cur = q.front();
                q.pop();
                for(int j = 0; j < cur.size(); j++)
                    for(auto it = has.begin(); it != has.end(); it++){
                        string temp = cur;
                        temp[j] = *it;
                        if(visited.count(temp) == 0){
                            if(judgek(temp, 0, temp.size()-1, k))
                                return cnt;
                            visited.insert(temp);
                            q.push(temp);
                        }
                    }
            }
        }
        return -1;
    }
};
```

3. 官方题解——动态规划

   我们用 $f[i][j]$ 表示对于字符串 S 的前 i 个字符，将它分割成 j 个非空且不相交的回文串，最少需要修改的字符数。在进行状态转移时，我们可以枚举第 j 个回文串的起始位置 i0，那么就有如下的状态转移方程：

   `f[i][j] = min(f[i0][j - 1] + cost(S, i0 + 1, i))` 其中 cost(S, l, r) 表示将 S 的第 l 个到第 r 个字符组成的子串变成回文串，最少需要修改的字符数。cost(S, l, r) 可以通过双指针的方法求出

   上述的状态转移方程中有一些边界情况需要考虑，例如

   - 只有当 i >= j 时，$f[i][j] $的值才有意义，这是因为 i 个字符最多只能被分割成 i 个非空且不相交的字符串，因此在状态转移时，必须要满足 i >= j 且 i0 >= j - 1

- 当 j = 1 时，我们并不需要枚举 i0，这是因为将前 i 个字符分割成 j = 1 个非空字符串的方法是唯一的。

```
class Solution {
public:
    int cost(string& s, int l, int r) {
        int ret = 0;
        for (int i = l, j = r; i < j; ++i, --j) {
            if (s[i] != s[j])
                ++ret;
        }
        return ret;
    }
    int palindromePartition(string& s, int k) {
        int n = s.size();
        vector<vector<int>> f(n + 1, vector<int>(k + 1, INT_MAX));
        f[0][0] = 0;
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= min(k, i); ++j) {
                if (j == 1)
                    f[i][j] = cost(s, 0, i - 1);
                else
                    for (int i0 = j - 1; i0 < i; ++i0)
                        f[i][j] = min(f[i][j], f[i0][j - 1] + cost(s, i0, i - 1));
            }
        }
        return f[n][k];
    }
};
```

4. 官方解法二——动态规划+预处理

方法一中的时间复杂度瓶颈在于 `cost()` 函数。在调用 `cost()` 函数之前，我们枚举了 `i` ，`j` 以及 `i0` ，因此它一共被调用了 $O(N^2K)$ 次。然而观察 `cost()` 函数本身的形式 `cost(S, l, r)` ，不同的 `(l, r)` 的数量只有 $O(N^2)$ 种，这说明在动态规划中，我们对 `cost()` 函数进行了大量的重复调用。因此我们可以预处理出所有的 `cost(S, l, r)` ，在后续调用 `cost()` 函数时，我们只需要 $O(1)$ 的时间便可以返回结果。

我们同样可以使用动态规划求出所有的 `cost(S, l, r)` 。记 `cost[l][r] = cost(S, l, r)` ，根据方法一中计算 `cost()` 函数的双指针方法，我们可以得到如下的状态转移方程：

```
cost[l][r] = cost[l + 1][r - 1],        if S[l] == S[r]
cost[l][r] = cost[l + 1][r - 1] + 1,    if S[l] != S[r]
cost[l][r] = 0,                         if l >= r
```

这是一个经典的区间动态规划，时间复杂度为 $O(N^2)$。在预处理出所有的 `cost(S, l, r)` 后，下一步使用动态规划计算 `f[i][j]` 的时间复杂度就从 $O(N^3K)$ 降低至 $O(N^2K)$。

```
class Solution {
public:
    int palindromePartition(string& s, int k) {
        int n = s.size();
        vector<vector<int>> cost(n, vector<int>(n));
        for (int span = 2; span <= n; ++span) {
            for (int i = 0; i <= n - span; ++i) {
```

```
                int j = i + span - 1;
                cost[i][j] = cost[i + 1][j - 1] + (s[i] == s[j] ? 0 : 1);
            }
        }

        vector<vector<int>> f(n + 1, vector<int>(k + 1, INT_MAX));
        f[0][0] = 0;
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= min(k, i); ++j) {
                if (j == 1)
                    f[i][j] = cost[0][i - 1];
                else
                    for (int i0 = j - 1; i0 < i; ++i0)
                        f[i][j] = min(f[i][j], f[i0][j - 1] + cost[i0][i - 1]);
            }
        }
        return f[n][k];
    }
};
```

5. 他人优化

$f[i][j]$ 只和 $f[1...i][j-1]$有关，所以其实不需要开两维的，反向遍历可以省掉第二维

```
int palindromePartition(const string &s, int k) {
        auto n = int(s.length());
        vector<vector<int>> mm(n + 1, vector<int>(n + 1));
        for (auto c = 0; c < n * 2; c++) {
            auto d = 0, i = c / 2, j = (c + 1) / 2;
            for (; 0 <= i && j < n; i--, j++)
                mm[i][j + 1] = d += s[i] != s[j];
        }
        vector<int> dp(n + 1, 0x3fffffff); dp[0] = 0;
        for (auto t = 1; t <= k; t++)
            for (auto j = n; j >= 0; j--)
                for (auto i = 0; i < j; i++)
                    dp[j] = min(dp[j], dp[i] + mm[i][j]);
        return dp[n];
    }
```