by Nathan Pinsker

Intuitively, the larger our rectangles are, the more likely they are to overlap other rectangles. Since we want to maximize the number of rectangles that can have been drawn first, we want as few overlaps as possible, since knowing that two rectangles overlap means that one of them cannot possibly have been drawn first. This means it is always to our advantage to assume, when we can, that an overlap doesn't occur, and so we will try to assume that each rectangle is as small as it can possibly be. In particular, for each rectangle of a certain color, we will always assume that the leftmost grid square of that color is its left border, the topmost grid square of that color is its top border, etc.

Now that we know the bounds of our rectangles, it is very straightforward to determine whether two rectangles overlap. But when two rectangles overlap, how do we know which one can be on top? Can either of them be on top? Consider the area formed when two rectangles of colors C and D overlap. Clearly it cannot contain squares of both colors C and D. If it contains either C or D, then the corresponding rectangle of that color must have been painted later. (Otherwise, the later rectangle would have painted over the color that shows up.) If it contains neither, then we can assume either rectangle was painted first, as we have no way to tell the difference.

This means that a rectangle R could have been painted first if, and only if, there is no other rectangle S such that there's a grid square of color R within the area of S.

This problem might seem to call for a two-dimensional segment tree at this point, but there's a significantly easier way. We can keep two-dimensional prefix sums representing the bounds of each rectangle to quickly compute, for any grid square, how many rectangles have been drawn over that square. Once we've done so, we iterate over every grid square. If a certain color appears at that square, and we calculate that two or more rectangles have been drawn at that square, then we "invalidate" that color -- it cannot possibly have been drawn first, as it's showing up over another rectangle.

Below is Brian Dean's solution. At the top-left and bottom-right grid squares of each rectangle, he increases the value of `P` at that square by 1, and at the bottom-left and top-right squares, he decreases the value of `P` by 1. Then he computes the array `A` as the prefix sums of that array. Each value of `A` will also be equal to the number of rectangles located at that point -- if you aren't sure why this is true, it's a good exercise to work out for yourself.

```
#include <iostream>
#include <fstream>
#define MAX_N 1000
#define MAX_C (MAX_N*MAX_N)
using namespace std;

int upper[MAX_C+1], lower[MAX_C+1], leftside[MAX_C+1], rightside[MAX_C+1];
int N, total, art[MAX_N+1][MAX_N+1], count[MAX_C+1];
int P[MAX_N+1][MAX_N+1], A[MAX_N+1][MAX_N+1];

int main(void)
{
  ifstream fin("art.in");
  ofstream fout("art.out");
  fin >> N;
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      fin >> art[i][j];
  for (int i=1; i<=N*N; i++)
    upper[i] = leftside[i] = N;
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++) {
      int c = art[i][j];
      if (c > 0) {
        if (count[c]==0) total++;
        count[c]++;
        upper[c] = min(upper[c], i+1);
        lower[c] = i+1;
        leftside[c] = min(leftside[c], j+1);
        rightside[c] = max(rightside[c], j+1);
      }
    }

  if (total==1) fout << N*N-1 << "\n";
  else {
    int answer = N*N-total;
    for (int c=1; c<=N*N; c++)
      if (c>0 && count[c]>0) {
        P[lower[c]][rightside[c]]++;
        P[lower[c]][leftside[c]-1]--;
        P[upper[c]-1][rightside[c]]--;
        P[upper[c]-1][leftside[c]-1]++;
      }
    for (int j=1; j<=N; j++)
      A[N][j] = P[N][j];
    for (int i=N-1; i>=1; i--) {
      A[i][N] = A[i+1][N] + P[i][N];
      for (int j=N-1; j>=1; j--)
        A[i][j] = A[i+1][j] + A[i][j+1] - A[i+1][j+1] + P[i][j];
    }
    for (int i=1; i<=N; i++)
      for (int j=1; j<=N; j++) {
        int c = art[i-1][j-1];
        if (c>0 && count[c]>0 && A[i][j]>=2) count[c] = 0;
      }
    for (int c=1; c<=N*N; c++)
      if (count[c]>0) answer++;
    fout << answer << "\n";
  }
  return 0;
}
```