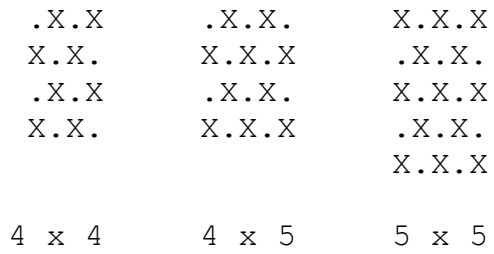
Code Jam 2015 - Round 1B

Analysis: Noisy Neighbors

Small number of tenants, N <= ceil(R * C / 2)

For N <= ceil(R * C / 2), we can put the tenants in a checkerboard pattern inside the R x C apartment complex and get the minimum possible unhappiness score of zero. See the following examples:



The left figure shows a 4 x 4 complex where we can place up to 8 tenants to apartments marked by â€˜Xâ€™ and get zero unhappiness score (an empty apartment is marked by â€˜.â€™). Observe that no two occupied apartments share a wall. Similarly for the middle figure, we can place up to 10 tenants and get zero unhappiness score.

With either R or C even (the left and middle figures), exactly half of the apartments can be rented. However, when both R and C are odd, two checkerboard patterns are possible. We should pick the checkerboard pattern with larger size. So for the right figure, we pick the checkerboard that has 13 â€˜Xâ€™ marks, instead of the other checkerboard with only 12 â€˜Xâ€™ marks.

Large number of tenants, N > ceil(R * C / 2)

For N > ceil(R * C / 2), some level of unhappiness will be present, since we must place at least one pair of tenants next to each other. Instead of starting with an empty building and adding tenants, it is easier to solve this case by starting with a building completely filled with tenants, and then removing K = R*C - N of them. We now need to determine which K tenants to remove in order to decrease the unhappiness score by as much as possible.

Letâ€™s first solve the smaller cases where R = 1 or C = 1. For those cases, we can always remove up to K tenants in such a way that each removal decreases the unhappiness score by 2, which is the maximum possible, and thus is optimal. To show this, look at the two possible examples below where C is even and C is odd:

For even C, K is at most C / 2 - 1 since N > ceil(R * C / 2). Thus for even C, it is always possible to remove tenants using the pattern shown on the left figure (marked by â€˜Xâ€™) to guarantee that each removal decreases the unhappiness score by 2 (i.e., the maximum possible decrease).

For odd C, K is at most (C - 1) / 2. In this case, it is always possible to remove tenants using the pattern shown on the right figure (marked by â€˜Xâ€™) to guarantee that each removal decreases the unhappiness score by 2 (i.e., the maximum possible decrease).

To get some idea for solving a general case where R and C are at least 2, letâ€™s observe the four buildings below with the checkerboard pattern. The cells marked by a number represent the apartments where the tenants may be removed. The numbers represent the amount of unhappiness each tenant is contributing to the building, or equivalently, the decrease in unhappiness that will occur if those tenants were removed:

```
 .3.2     .3.3.     2.3.2     .3.3.
 3.4.     3.4.3     .4.4.     3.4.3
 .4.3     .4.4.     3.4.3     .4.4.
 2.3.     2.3.2     .4.4.     3.4.3
                    2.3.2     .3.3.

4 x 4     4 x 5     5 x 5     5 x 5
```

We can devise an optimal K-tenant-removal strategy (i.e., remove K tenants such that the total unhappiness score is minimal) as follows:

- For K <= (R - 2) * (C - 2) / 2, we can always remove K tenants from the inner building (at the positions marked by â€˜4â€™ in the figures) where each tenant removal decreases the unhappiness score by 4, which is the maximum decrease we can get, and thus it is optimal.
- For K > (R - 2) * (C - 2) / 2, after removing all the tenants at positions marked by â€˜4â€™, we start removing more tenants at the sides of the building marked by â€˜3â€™ (each tenant removal at these positions decreases the unhappiness score by 3). If all tenants at the sides of the building have been removed but we still need to remove more tenants (i.e., the total number of removed tenants has not reached K yet), remove more tenants at the corners of the building marked by â€˜2â€™ (each tenant removal at these positions decreases the unhappiness score by 2). It is guaranteed that K tenants can be removed by now, since K is at most R * C / 2.

Note that for odd R and odd C, there are two possible checkerboard patterns. Both of them must be considered, and we pick the one that yields the minimum unhappiness score.

Why does this strategy work?

Each removal can reduce unhappiness by at most 4. Below is a 5 x 5 building where the number in each apartment cell is the amount of unhappiness score reduction if the tenant at that cell location is removed:

```
23332
34443
34443
34443
23332
```

We can place at most ceil((R-2) * (C-2) / 2) 4s, so we cannot do better than to place this many 4s, and for the remainder place 3s. So we can put an upper bound U on the amount of unhappiness reduction we can achieve — U <= 4 * K for K <= ceil((R-2) * (C-2) / 2), and U <= 3 * K + ceil((R-2) * (C-2) / 2) otherwise.

Consider the removal strategy using this checkerboard pattern (let's call this *pattern1*):

```
2.3.2
.4.4.
3.4.3
.4.4.
2.3.2
```

This achieves U exactly when K <= floor(R * C / 2) - 3 for R and C odd, and K <= floor(R * C / 2) - 2 when one of R and C are even, so this strategy is optimal for these cases.

For the remaining cases, this strategy achieves either U-1 and U-2, because we need to place some 2s. Since this strategy uses the maximal number of 4s, and the maximal number of 3s possible when using the maximal number of 4s, the only possible way to improve upon it would be to use 1 fewer 4 in order to use more 3s.

It is indeed possible to do this to improve the case K = floor(R * C / 2) - 1 from 4 total unhappiness to 3 total unhappiness, using the other checkerboard pattern (letâ€™s call this *pattern2*):

```
.3.3.
3.4.3
.4.4.
3.4.3
.3.3.
```

That is, if we were to use the checkerboard *pattern1*, removing K = 11 tenants reduces the unhappiness score by 5*4 + 4*3 + 2*2 = 36. On the other hand, *pattern2* reduces the unhappiness score by 4*4 + 7*3 = 37, by using 1 fewer 4s and more 3s.

On the other hand, *pattern1* is optimal is when K = 5, for example. In this case, *pattern1* reduces the unhappiness score more than *pattern2* since *pattern1* has 5 of 4s while *pattern2* only has 4 of 4s and would need to remove another 1 of 3s.

Sample implementation in C++:

```
#include <cassert>
#include <cstdio>
#include <algorithm>

using namespace std;

int T, R, C, N;

int remove_tenants(int &K, int max_remove, int remove_cost) {
  int removed = min(K, max_remove);
  K -= removed;
  return removed * remove_cost;
}

int get_score(int all, int corners, int inners) {
  int sides = all - corners - inners;
  int K = R * C - N;
  int unhappiness = R * C * 2 - R - C;
  unhappiness -= remove_tenants(K,  inners, 4);
  unhappiness -= remove_tenants(K,   sides, 3);
  unhappiness -= remove_tenants(K, corners, 2);
  assert(K == 0);
  return unhappiness;
}

int min_unhappines() {
  // Guaranteed zero unhappiness.
  if (N <= (R * C + 1) / 2) return 0;
```

```
  if (R == 1) {
    int K = R * C - N;
    int unhappiness = C - 1;
    int remove_cost = 2;
    return unhappiness - K * remove_cost;
  }

  if (R % 2 == 1 && C % 2 == 1) {
    // 2.3.2
    // .4.4.
    // 3.4.3
    // .4.4.
    // 2.3.2
    int pattern1 = get_score(
      (R * C + 1) / 2,  // Max #tenants that can be removed.
      4,  // #tenants at the corners of the building.
      ((R-2) * (C-2) + 1) / 2  // #tenants at inner building.
    );

    // .3.3.
    // 3.4.3
    // .4.4.
    // 3.4.3
    // .3.3.
    int pattern2 = get_score(
      (R * C) / 2,  // Max #tenants that can be removed.
      0,  // #tenants at the corners of the building.
      ((R-2) * (C-2)) / 2  // #tenants at inner building.
    );

    return min(pattern1, pattern2);
  }

  // .3.2      2.3.      2.3.2
  // 3.4.  or  .4.3  or  .4.4.
  // .4.3      3.4.      3.4.3
  // 2.3.      .3.2      .3.3.
  return get_score(
    (R * C + 1) / 2,  // Max #tenants that can be removed.
    2,  // #tenants at the corners of the building.
    ((R-2) * (C-2) + 1) / 2  // #tenants at inner building.
  );
}

int main() {
  scanf("%d", &T);
  for (int TC = 1; TC <= T; TC++) {
    scanf("%d %d %d", &R, &C, &N);
    if (R > C) swap(R, C);
    printf("Case #%d: %d\n", TC, min_unhappines());
  }
}
```