

Analysis: Counter Culture

For the small dataset, it suffices to use a breadth-first search to find the minimum number of moves required to generate all numbers from 1 to 10^6 . You will never want to construct a number larger than 10^6 and later flip it to get a smaller number, so you only need to consider 10^6 states.

For the large dataset, the following solution works. The key ideas are:

- You should first build the numbers 10, 100, 1000, ... until you get a power of 10 with the same number of digits as N, and then build N.
- You should make at most one flip while the number is at a given number of digits.

Our algorithm works in two parts.

Part 1: Get to the right number of digits as fast as possible.

To get from a 1 followed by X 0s to a 1 followed by X+1 0s: first count until the right half of the number is filled with 9s. (When the length is odd, make the left half shorter than the right half.) Then flip, then count until the number is all 9s. Then add 1. (To get from 1 to 10, obviously we don't need to flip.)

Part 2: Either count directly to the answer, or do some counting, flip, and then do some more counting, whichever is faster.

Once we're at the right number of digits, we use a similar algorithm to produce N: count until the right half looks like the left half of N in reverse, then flip, then count up to the target. For example, to get from 100000 to 123456:

- count to 100321
- flip to get 123001
- count to 123456.

If the left half is just a 1 followed by 0s, we can skip the first two steps and just count to N.

When the right half of N is all zeroes the above method doesn't work, because after we flip, the right half ends with 1. Instead, we make the right half look like the left half of N-1, flip, then count up to N. For example, to get from 100000 to 300000:

- count to 100992
- flip to get 299001
- count to 300000.

However, as before, if the left half of N-1 is a 1 followed by 0s, we can skip the first two steps. For example, to get from 100000 to 101000, it's best to count up directly.

Why do we need to do at most one flip? The goal of the flip is to allow us to reach the left half of the number as quickly as possible. Raising the right half to the desired value is better done by directly counting the number up. In other words, there is no added value in doing multiple flips.

Sample implementation in C++:

```

#include <cstdio>
#include <cstdlib>

#include <string>
#include <algorithm>

using namespace std;

long long p10[10]; // p10[i] == 10^i

bool is_1_followed_by_0s(string S) {
    reverse(S.begin(), S.end());
    return atoi(S.c_str()) == 1;
}

int solve(long long N) {
    if (N < 10) return N; // Trivial case.

    char X[20]; sprintf(X, "%lld", N);
    string S = X;
    int M = S.length(); // Number of digits of N.

    // Starts from 1.
    int ans = 1;

    // Part 1: from 1, get to the M digits as fast as possible.
    for (int d = 1; d < M; d++) {
        // For digits = 7, it starts from 7 digits: 1000000
        ans += p10[(d + 1) / 2] - 1; // Count up 9999: 1009999
        if (d > 1) ans++;           // Flip once: 9999001
        ans += p10[d / 2] - 1;      // Count up 999: 10000000
    }

    // Part 2:

    // Split N into two halves. For example N = "1234567"
    string L = S.substr(0, M / 2); // L = "123"
    string R = S.substr(M / 2);    // R = "4567"

    // Handles the case where the right half is all zeroes.
    if (atoi(R.c_str()) == 0) return solve(N - 1) + 1;

    // Special case: Count directly to the answer.
    if (is_1_followed_by_0s(L))
        return ans + atoi(R.c_str());

    // Count until the right half looks like the left half of N
    // in reverse. In this case, count from 1000000 to 1000321.
    reverse(L.begin(), L.end());
    ans += atoi(L.c_str());

    // Flip 1000321 to 1230001.
    ans++;

    // Count up 4566 to the target from 1230001 to 1234567.
    ans += atoi(R.c_str()) - 1;
}

```

```
    return ans;
}

int main() {
    p10[0] = 1;
    for (int i = 1; i < 10; i++)
        p10[i] = p10[i - 1] * 10;

    long long T, N;
    scanf("%lld", &T);
    for (int TC = 1; TC <= T; TC++) {
        scanf("%lld", &N);
        printf("Case #%d: %d\n", TC, solve(N));
    }
}
```

Analysis: Noisy Neighbors

Small number of tenants, $N \leq \text{ceil}(R * C / 2)$

For $N \leq \text{ceil}(R * C / 2)$, we can put the tenants in a checkerboard pattern inside the $R \times C$ apartment complex and get the minimum possible unhappiness score of zero. See the following examples:

.X.X	.X.X.	X.X.X
X.X.	X.X.X	.X.X.
.X.X	.X.X.	X.X.X
X.X.	X.X.X	.X.X.
		X.X.X

4 x 4 4 x 5 5 x 5

The left figure shows a 4×4 complex where we can place up to 8 tenants to apartments marked by $\hat{\sim}X\hat{\sim}$ and get zero unhappiness score (an empty apartment is marked by $\hat{\sim}.\hat{\sim}$). 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 $\hat{\sim}X\hat{\sim}$ marks, instead of the other checkerboard with only 12 $\hat{\sim}X\hat{\sim}$ marks.

Large number of tenants, $N > \text{ceil}(R * C / 2)$

For $N > \text{ceil}(R * C / 2)$, 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.

Let'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:

.X.X..	.X.X.X.
even $C=6$	odd $C=7$

For even C , K is at most $C / 2 - 1$ since $N > \text{ceil}(R * C / 2)$. Thus for even C , it is always possible to remove tenants using the pattern shown on the left figure (marked by $\hat{\sim}X\hat{\sim}$) 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 $\hat{\sim}X\hat{\sim}$) 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, let'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 $K \leq (R - 2) * (C - 2) / 2$, we can always remove K tenants from the inner building (at the positions marked by $\sim 4 \sim$ 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 $\sim 4 \sim$, we start removing more tenants at the sides of the building marked by $\sim 3 \sim$ (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 $\sim 2 \sim$ (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 5×5 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 $\text{ceil}((R-2) * (C-2) / 2)$ 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 — $U \leq 4 * K$ for $K \leq \text{ceil}((R-2) * (C-2) / 2)$, and $U \leq 3 * K + \text{ceil}((R-2) * (C-2) / 2)$ 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 $K \leq \text{floor}(R * C / 2) - 3$ for R and C odd, and $K \leq \text{floor}(R * C / 2) - 2$ 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 $K = \text{floor}(R * C / 2) - 1$ from 4 total unhappiness to 3 total unhappiness, using the other checkerboard pattern (let'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());
    }
}

```

Analysis: Hiking Deer

Some programming contest problems have their origins in everyday life. This year, the Qualification Round's "Standing Ovation" was written during a concert, and Round 1A's "Haircut" was written while waiting in a long line for a sandwich at Google's Go Cafe. You may not be surprised to learn that this problem was inspired by a circular hike on a day with an unusually large number of hikers on the trail!

Let H be the total number of hikers. One important observation is that the answer can't be larger than H , since no matter what the hikers' initial positions and speeds are, Herbert can just go around so fast that no hiker gets a chance to move very far, so he encounters each of them once.

Another key observation is that allowing Herbert to wait or vary his speed makes no difference to the answer. If we know the time he arrives at the end, we can put a lower bound on the answer based on the number of hikers he must have overtaken and the number of times hikers must have overtaken him. This lower bound is exactly the answer we get by having Herbert move at a constant speed, so no strategy of waiting or changing speed can improve on it.

So now, we will find a finish time that minimizes the number of encounters.

Consider the case of a single hiker. As the trail is circular and the hiker walks endlessly, the hiker repeatedly reaches Herbert's starting point. Call the time at which Herbert finishes his hike X , and the times when the hiker reaches Herbert's starting point T_1, T_2, T_3 , etc.

- If $X \leq T_1$, then there is one encounter with the hiker as Herbert passes.
- If $T_1 < X < T_2$, then there are no encounters with the hiker.
- If $T_2 \leq X < T_3$, then the hiker passes Herbert once.
- If $T_3 \leq X < T_4$, then the hiker passes Herbert twice.
- etc.

So as we increase Herbert's finish time, there are multiple "events" where the number of encounters changes. It decreases by 1 for the first event, then it increases by 1 for each event after that.

Now when we have multiple hikers, there is a series of events for each of them that can increase or decrease the total number of encounters.

There will never be a reason for Herbert to take so long that we see more than H events for a single hiker, because then the number of encounters with that single hiker will be at least H . So a solution that will work for the small datasets is to make a sorted list of the first H events for all of the hikers, and then find the optimal number of encounters in this way:

- Initialize the number of encounters to H .
- For each event, in order of time, subtract 1 from the number of encounters if it is the first event for the hiker, otherwise add 1.

The answer is the minimum number of encounters for any time.

For the large dataset, we need a more efficient solution. Note that there are only H events that subtract 1 from the number of encounters, and the rest add 1. So once we have processed $2H$ events, we will never find a time with fewer than H encounters, so we can stop searching.

We still need to avoid storing H^2 events. To do this, we can maintain a priority queue of events. We initialize the queue with one event for each hiker -- the event for that hiker's T_1 . Whenever we process an event, we replace that event on the queue with the next event for that hiker. In this way we only need $O(H)$ memory, and $O(H \log H)$ time.

One subtlety to note is that if multiple hikers reach Herbert's starting point at the same time, we must process the events that add an encounter (those for a hiker's T_2, T_3 , etc.) before the events that subtract an encounter (those for a hiker's T_1) since the number of encounters only really decreases once Herbert's finish time is *larger* than T_1 .

52 people successfully solved the large input for this problem during the contest. See, for example, Belonogov's solution, which you can download from the scoreboard.