

Analysis: Standing Ovation

What kind of shyness level friends should we invite?

Since we can invite any friend with any shyness level, the problem seems really complicated. However, if you think about it you will realize that inviting friends with shyness level 0 is optimal and makes this problem simpler. Friends with shyness level 0 are always better than those with other shyness levels because they always stand up and clap, therefore helping all other audience members who have shyness level greater than 0. This kind of choice is greedy but makes sense as for any scenario: if you can invite friends with shyness level greater than 0 and solve the problem, then you can always replace them with friends whose shyness level is 0 and still solve the problem.

How many friends should we invite?

For audience members with shyness level k , to meet their needs, we must have at least k people who already stood up before them. These include both the audience members whose shyness level is less than k and the friends we invited (with shyness level 0). Assuming that there are t audience members who stood up already, the number of friends we need to invite is $\max(k - t, 0)$. This provides us with an algorithm. Let's say we are given an array of audience members for which the k^{th} entry contains the total audience members of shyness level k . Now, for each shyness level, we compute the number of friends we need to invite to get the audience members of that level to stand up and clap, and we record the maximum value out of all such values. This max value is the minimum number of friends we need to invite to let every audience member stand up and clap for the prima donna.

Below is a sample implementation in Python:

```
for tc in range(input()):
    smax, string = raw_input().split()
    t = 0
    min_invite = 0
    for k in range(int(smax) + 1):
        min_invite = max(min_invite, k - t)
        t += int(string[k])
    print "Case #%d: %d" % (tc + 1, min_invite)
```

Because the input of audience members is already given in increasing order, we only need to walk through this array once and compute the result. Since `min_invite` is initially 0, $\max(k - t, 0)$ is redundant (i.e., $k - t$ is sufficient). The overall time complexity is $O(S_{\max})$.

Analysis: Infinite House of Pancakes

As the head server, you have two choices for each minute:

1. **eat**: you do nothing and let every diner with a non-empty plate eat one pancake from his or her plate.
2. **move**: you ask for the diners' attention for a *special* minute, choose a diner with a non-empty plate, and move some number of pancakes from that diner's plate to another diner's (empty or non-empty) plate.

Imagine that you, as the head server, have planned a strategy to move the pancakes. Let's call it **strategy A**. In this strategy, you pick a successive pair of minutes t and $t+1$ such that at minute t you choose **eat** and at minute $t+1$ you choose **move**. As a natural philosopher, you start to question your decision, "What if I swap my plan at minute t with my plan at minute $t+1$, i.e. **move** pancakes at minute t (instead of $t+1$) and **eat** at minute $t+1$ (instead of t)?". Out of curiosity, you swap your plan at minute t and $t+1$ and calculate the breakfast end time. Surprisingly, breakfast does not take longer. After that, you start to find several other successive pairs of minutes with the same property and do other swaps. You find that breakfast never takes longer than if you strictly follow **strategy A**. Surprisingly, on certain swaps, breakfast even ends earlier than **strategy A**!

Of course you become more curious! You start to observe what happened when you did those swaps. If the plan at minute t is **eat** and the plan at minute $t+1$ is **move**, you notice that all pancakes that were eaten at time t before you did the swap were also eaten at time $t+1$ after swapping. Hence, the number of pancakes that were eaten after time $t+1$ does not decrease, which implies that breakfast will not take longer. After you swap the plan at t with $t+1$, you notice that several plates that were empty at time t might become non-empty at time $t+1$ after swapping (and hence, the number of pancakes that were eaten after time $t+1$ might increase by one).

"Whoa! That means if I swap **<eat, move>** pairs to **<move, eat>**, the amount of pancakes that were eaten at time t before the swap is either less than or equal to the amount of pancakes that were eaten at time $t+1$ after the swap. Hence, such swaps are always good (i.e. it will not make the breakfast time longer but might make the breakfast time shorter)!"

Excited with your finding, you grab a whiteboard and start to swap every successive pairs of **<eat, move>** to be **<move, eat>**, until none of the **<eat, move>** pairs are left. You look at your revised strategy.

"**Move, move, move, eat, eat, eat, eat, eat**" you read out loudly. "Of course! *The only strategy without a <eat, move> pair is the strategy in which we do all the moves at the beginning, and always eat after that.* If we look at this carefully, swapping pairs of elements that are in the wrong order (in this case **eat** then **move**) until none are left will actually *sort* the strategy. This is basically doing [bubble sort](#) on **strategy A**!"

Knowing that you are going to get a bonus from your boss for ending breakfast early, you quickly present this solution to your boss. Your boss is not convinced. He replies without showing much interest, "Well, I understand that **moving** pancakes at the beginning will always lead to an optimal solution. But what is more important is that for every **move**, do you know how you are going to move the pancakes?" You quickly answer, "At least I know that we can always move pancakes to empty plates! There are infinitely many of them, and we do not have any

reason to let diners with non-empty plates to eat more than is needed. We cannot even wait for them to finish their existing pancakes!"

Your boss stops reading emails and starts to pay attention to your explanation. "How will you pick the plates to **move** pancakes from?" he asks. "Perhaps from the customer with the most pancakes on their plate? They need the most help to finish their pancakes," you reply.

"You mean we can end breakfast earlier if we keep moving several pancakes from the plates with most pancakes to empty plates, and stop moving after several minutes?" he enthusiastically says. "Congratulations! You get a big bonus and extra time off!"

You are extremely happy. You gather all the kitchen staff and explain the new strategy... of course after bragging a bit on social media! A new intern staff raises her hand and asks, "When should we stop **moving** pancakes? Also, how many pancakes should we move every time?"

The whole kitchen goes silent. Everyone is looking at you expecting answers. Breakfast will start in half an hour. "Quick, think of something useful," you tell yourself. Your mind feels like it is exploding!

"Erm, honestly I don't know. Let's try to simulate breakfast to get some insights. Please bring me two plates of pancakes with **15** and **17** pancakes respectively. Ah, don't forget to bring me several empty plates too," you order one of your staff.

You start to simulate your new strategy multiple times with different amount of pancakes to be moved and stop moving at different times. Suddenly, your new intern says to you, "Look! You take **10** pancakes from the **plate 1** and put it on an **empty plate 3**. Later, you picked **3** pancakes from **plate 3** and moved it to another **empty plate 4**. Instead of doing that, why didn't you move **7** pancakes from first plate to an **empty plate 3**, and then move **3** pancakes from first plate to another **empty plate 4**? That way, you can always move pancakes from plates that are initially non-empty."

"I see," you say, "That means we can take half the pancakes from the plate with largest amount of pancakes, then move it to an empty plate, then pick another plate with the largest amount of pancakes and do the same thing multiple times..."

She replies quickly, "Your strategy is bad. Imagine that you had a plate of **9** pancakes. If we use your strategy, the best we can do is to split the plate into plates with **4** and **5** pancakes, and let the diners eat them. It costs us **6** minutes to end breakfast. But if you split the plate into three plates with **3** pancakes each (using **2 moves**), we can finish breakfast in only **5** minutes!"

"I have a suggestion," she continues, "If you expect the customer to finish their pancakes **x** minutes after you stop moving pancakes, any strategy that satisfies this will be equivalent to repeatedly moving at most **x** pancakes from every initially non-empty plate to an empty plate until the number of pancakes on the initially non-empty plate becomes no more than **x**."

"And if you always move exactly **x** pancakes, for a plate with **P_i** pancakes you need only **M(P_i)=ceil(P_i/x)-1** moves until the number of pancakes in the plate is no more than **x**. You cannot do less than that! Overall, we are going to move **sum of M(P_i)** times, where **P_i** is the number of pancakes on **plate i**. That is the minimum amount of moves that we can do for the given **x**! We can try all possible values of **x** to find the optimal **x** that has earliest breakfast end time!" Everyone gives her a standing ovation.

"Let's summarize our strategy. First of all, we fix a number **x** to be the number of minutes that we expect breakfast to end in after we stop moving pancakes. After that, we pick a plate with more than **x** pancakes, take **x** pancakes from that plate and

move the pancakes to an empty plate. We keep doing that until all plates has at most x pancakes, then we let the customers eat their pancakes and breakfast ends the earliest for that x value! If we move **sum of $M(P_i)$** times, in total, the breakfast ends exactly after **$x + \text{sum of } M(P_i)$** minutes," you say. "And we can try all possible values of x , since the amount of pancakes cannot be more than **1000**. The complexity of the algorithm is **$O(D \cdot M)$** , where **D** is the number of diners and **M** is the maximum number of pancakes. This is fast enough to solve our problem. We can use a spreadsheet to..."

"Certainly, but I have prepared for this," says your intern. She opens her laptop and types some really short functions, of course in **C++**. "Why? Because **C++** is cool!"

```
// Get the minimum possible breakfast end time, given
// P[i] is the number of pancakes of diner i initially.
int f(const vector<int>& P) {
    const int max_pancakes = *max_element(P.begin(), P.end());
    int ret = max_pancakes;
    for (int x = 1; x < max_pancakes; ++x) {
        int total_moves = 0;
        for (const int Pi : P) {
            // (Pi - 1) / x is equivalent to M(Pi),
            // which is ceil(Pi / x) - 1
            total_moves += (Pi - 1) / x;
        }
        ret = min(ret, total_moves + x);
    }
    return ret;
}
```

"Whoaa! Run it, run it!" you exclaim.

"Hold on, hold on. Although the algorithm above is fast enough to solve our problem, I have an even faster algorithm. Notice that the list of **ceil(a/1)**, **ceil(a/2)**, ... only changes values at most **$2 \cdot \sqrt{a}$** times. For example, if **a=10**, the list is: **10, 5, 3, 3, 2, 2, 2, 2, 2, 1, 1, ...**. That list only changes value **4** times which is less than **$2 \cdot \sqrt{10}$** ! Therefore, we can precompute when the list changes value for every diner in only **$O(D \cdot \sqrt{M})$** . We can keep track these value changes in a table. For example, if **$P_i=10$** , we can have a table **T_i** : **10, -5, -2, 0, -1, 0, 0, 0, 0, -1, 0, ...**.

Notice that the prefix sum of this table is actually: **10, 5, 3, 3, 2, 2, 2, ...**. More importantly, this table is sparse, i.e. it has only **$O(\sqrt{M})$** non-zeroes. If we do vector addition on all **T_i** , we can get a table where every entry at index **x** of the prefix sum contains sum of **ceil(P_i/x)**. Then, we can calculate sum of **ceil(P_i/x)-1** in the code above by subtracting the **x^{th}** index of the prefix sum with the number of diners. Hence, only another **$O(M)$** pass is needed to calculate candidate answers, which gives us **$O(D \cdot \sqrt{M}) + M$** running time. A much faster solution!"

"One more thing, we cannot do binary search or ternary search, at least in trivial ways, on a function that maps x to its minimum breakfast end time as the function can have multiple minimas. e.g. for **2** diners with **9** pancakes each, the function forms the following mappings: **(1,17), (2,10), (3,7), (4,8), (5,7), (6,8), (7,9), (8,10), (9,9)**."

"I don't need that! The previous algorithm is fast enough. Please run it," you say impatiently.

"Sigh. I won't tell you how to code the faster solution then. The answer is..." Everyone gasps while the program blinks. "... **42**."

Analysis: Dijkstra

We can solve this problem with the help of the following two insights. First, the product of the whole string must equal -1 ($i \hat{=} j \hat{=} k$). Second, we only need a small number of copies of the input string to check for a solution. One way to do this is by finding the shortest prefix of the whole string that reduces to i and the shortest subsequent prefix that reduces to j . If we find such two substrings, there is a solution because the rest of the string reduces to k thanks to the associative property. The rest of this editorial explains the two insights and provides a sample implementation.

No solution if the whole string cannot be reduced to -1

Suppose the string **S** is the whole concatenated string formed by repeating the given input string **X** times. If we can break the string **S** into three non-empty substrings **A**, **B**, **C** where $A + B + C = S$ such that **A** reduces to i , **B** reduces to j , and **C** reduces to k , then the string **S** reduces to string "ijk", which then reduces to -1 . Therefore, if the string **S** cannot be reduced to -1 , then there is no solution. Reducing the string **S** can be done by simply multiplying all the characters in **S** into one value.

If the string **S** can be reduced to -1 , it doesn't mean that there is a solution for **S**. There are many strings (e.g., "ii", "jj", etc.) that reduces to -1 but do not form a concatenation of three substrings that reduce to i , j , and k , respectively.

The first two substrings must reduce to i and j , respectively

From now on, we only consider whole strings that reduce to -1 . To determine whether a string **S** can be broken into three non-empty substrings **A**, **B**, **C** where each reduces to i , j , k respectively, we only need to find the first two substrings. The last substring is guaranteed to reduce to k since the whole string reduces to -1 .

(There are other alternatives. For example, we can find the shortest prefix and the shortest suffix that reduce to i and k , respectively. If the prefix does not overlap with the suffix, then we can reduce the rest to j . Care must be taken as the multiplication operator is not commutative. Exercise: Can the prefix-suffix pair actually overlap while the whole string can still be reduced to -1 ? Answer: No.)

To find the first substring, we start from the first character of the string **S** and start multiplying it with the next characters and so on until we get the value i . Afterward, we repeat the procedure from the current position until we get the value j . The rest of the string is guaranteed to be non-empty and reduce to k . The complexity of finding the first and the second substrings is $O(L * X)$, since we need to scan the whole string **S** of length $L * X$.

With these insights, we can solve the small input in $O(L * X)$. Below is a sample implementation in Python:

```
M = [[ 0, 0, 0, 0, 0 ],
      [ 0, 1, 2, 3, 4 ],
      [ 0, 2, -1, 4, -3 ],
      [ 0, 3, -4, -1, 2 ],
      [ 0, 4, 3, -2, -1 ]]
```

```
def mul(a, b):
```

```

    sign = 1 if a * b > 0 else -1
    return sign * M[abs(a)][abs(b)]

def multiply_all(S, L, X):
    value = 1
    for i in range(X):
        for j in range(L):
            value = mul(value, S[j])
    return value

def construct_first_two(S, L, X):
    i_value = 1
    j_value = 1
    for i in range(X):
        for j in range(L):
            if i_value != 2:
                i_value = mul(i_value, S[j])
            elif j_value != 3:
                j_value = mul(j_value, S[j])
    return i_value == 2 and j_value == 3

for tc in range(input()):
    L, X = map(int, raw_input().split())
    # maps 'i' => 2, 'j' => 3, 'k' => 4
    S = [(ord(v) - ord('i') + 2) for v in raw_input()]
    ok1 = multiply_all(S, L, X) == -1
    ok2 = construct_first_two(S, L, X)
    print "Case #%d: %s" % (tc + 1,
        "YES" if ok1 and ok2 else "NO")

```

The multiplication matrix M is defined as a 5×5 matrix where the first column and the first row are not used (for value 0). The second row and the second column is for an identity value 1. The third, fourth and fifth (rows and columns) represent i, j , and k , respectively, identical to the quaternion multiplication matrix.

Optimizations for the large input

The maximum whole string length is 10^{16} which is too large for an implementation of the algorithm above to finish within the time limit when it is executed in a today's computer. We need to optimize both of these functions: `multiply_all()` and `construct_first_two()`.

Optimizing the `multiply_all()` method

Observe that the whole string is formed by repeating the input string (of length L) X times, giving the complexity of $O(L * X)$. Since the multiplication operator is associative, we can reduce the input string into a single value before multiplying this value to itself $X - 1$ times. Thus, we can rewrite the `multiply_all()` method as follows:

```

def multiply_all(S, L, X):
    value = 1
    for i in range(L):
        value = mul(value, S[i])
    return power(value, X) # computes value^X

```

To quickly multiply the value with itself $X - 1$ times (that is, computing value^X), we can use the [exponentiation by squaring](#) technique which runs in $O(\log(X))$:

```
def power(a, n):
    if n == 1: return a
    if n % 2 == 0: return power(mul(a, a), n // 2)
    return mul(a, power(mul(a, a), (n - 1) // 2))
```

With this optimizations, the multiply_all() complexity is now $O(L + \log(X))$. Since L is at most 10000 and X is at most 10^{12} , the number of multiplication operations is at most 10040.

We can improve it further to $O(L)$. Observe that the multiplication values (to itself) will always repeat to the original value after 4 consecutive multiplications, thus we only need to do at most $X \bmod 4$ multiplications to compute value^X :

```
def power(a, n):
    value = 1
    for i in range(n % 4):
        value = mul(value, a)
    return value
```

Optimizing the call to construct_first_two()

To find the prefix, we start with an identity value 1 and multiply it with the value of the first position of the input string, and so on until we get a value i . Supposing X is sufficiently large, if we reach the end of the input string and haven't obtained the value i , we repeat this procedure for the next copy of the input string. At this point, the current value may be 1, j , k , -1 , $-i$, $-j$, or $-k$. If the current value is 1, we may as well stop here and declare no solution because continuing the multiplication will result in the same value 1 again. However, if the current value is not 1, we can continue the procedure.

We know from the previous section that multiplying a value to itself will repeat to its original value every 4 consecutive multiplication. Thus, if we don't encounter the value i after executing the procedure for 4 copies of the input string, it is impossible to construct the desired prefix. If we do encounter value i , then we can proceed to find the second substring where the same insight applies.

Thus, we can safely limit the call to construct_first_two() from X repeats:

```
ok2 = construct_first_two(S, L, X)
```

to $\min(8, X)$ repeats:

```
ok2 = construct_first_two(S, L, min(8, X))
```

This reduces the complexity of the construct_first_two() method from $O(L * X)$ to $O(L)$.

Therefore, the complexity of the overall algorithm is now $O(L)$ per test case.

Analysis: Ominous Omino

Introduction

There are various aspects to solving this problem. First off, there is one key observation about X-ominoes where $X \geq 7$ which we will cover shortly. Beyond that, we describe a brute force solution where we generate all possible X-ominoes, simulate Richard picking one of the X-ominoes, then simulate Gabriel placing that particular X-omino in all possible locations on the grid to determine if a winning configuration can be found. We also present an alternative solution that involves an exhaustive case-by-case analysis of input to determine the winner.

Trivial case where $X \geq 7$

There is a key observation for X-ominoes where $X \geq 7$. We point out that in such cases, it is impossible for Gabriel to win. How is that?

```
###  ####
#.#  #.##
##.  ###.
```

When $X \geq 7$, Richard can always choose a X-omino which has a hole in the middle as shown in the figure above (a 7-omino in the left, and a 10-omino in the right). This means that it is impossible for Gabriel to win using that X-omino at least once, as it is impossible to fill the hole in the middle with another X-omino.

Therefore in cases where $X \geq 7$, Richard will always win.

Available cells is a multiple of X

Another insight is the following: if $R \times C$ is not a multiple of X, then it is impossible for Gabriel to win. Why? The X-omino occupies X cells exactly, so if the total number of cells is not a multiple of X, it is impossible to cover all the cells using only X-ominoes. Therefore, in such cases, Richard will always win.

In fact, we can extend this principle. Let's imagine that we have already placed one X-omino in the $R \times C$ grid, and it results in the $R \times C$ grid being separated into two (or more) edge-connected regions. In the figure below, we use a 4-omino in a 2×6 .

```
.##...
##....
```

Since the sizes of the connected component (connected components can be found by using [flood fill](#)) of the blank areas represented as '.' are 1 and 7, and 1 is too small to fit while 7 is not a multiple of X, then it is not possible for Gabriel to win that particular configuration. Therefore we can say that if we ever arrive in a grid configuration where any of the remaining connected components size is not a multiple of X then we can say that Richard will win that particular configuration. If a connected blank area of size M is a multiple of X, it can be guaranteed that there is a way to place M/X X-ominoes to fill in the blank area. It can be proven by using exhaustive case-by-case analysis which is described in the next section.

Now, armed with the above knowledge, we proceed with describing a brute force solution. First, we describe a way to generate all possible X-ominoes. Then we describe the general brute

force strategy to test if Richard can pick a X-omino that will guarantee him a win, and if he is unable to do so then Gabriel will win.

Generating all possible X-ominoes

To generate all possible X-ominoes, we describe a recursive strategy. Let's start with a 1-omino. Well, for a 1-omino there is trivially only 1 possibility which is the following:

#

Similarly, we can generate the configurations for a 2-omino as follows:

#

How can we do this? We can build the X-ominoes recursively. We start with an empty board (let's say of size 20x20) then place a '#' in middle. Then we can recursively add a '#' adjacent to any of the existing '#' we have already placed, and stop when we have placed X '#' to create a X-omino. But you might have noticed that this recursive process will likely create a lot of duplicate X-omino configurations, e.g. in the 2-omino case, after placing the '#' in the middle, there are 4 possible placements of an adjacent '#' (labeled as 1-4 in the figure below).

.1.
4#2
.3.

This means that this recursive procedure will generate four 2-ominoes! But we know that there are only two 2-ominoes, as "4#" and "#2" are equivalent, and likewise for "1#" and "3#". Therefore, we can come up with a way to remove duplicate X-omino configurations if needed, which we leave as an exercise.

We can precompute all the X-ominoes for $1 \leq X \leq 6$. Also note that when generating X-ominoes this way, we will generate all X-ominoes under reflection and rotations which we show with an example below:

#. .# .## ##.
##. .##
.# #.

For our solution, it is fine to generate all such X-ominoes. Note that it is also feasible to generate all possible X-ominoes for $X \leq 6$ by hand. We cover this in the "Alternative Solution" section below.

Brute force strategy

Now, after we have generated all possible X-ominoes, we proceed with describing the brute force strategy. As mentioned earlier, if $X \geq 7$, we report that Richard will win, and also if $R \cdot C$ is not a multiple of X , we report that Richard will win.

In the brute force strategy, we simulate Richard picking each of the X-ominoes one-by-one, then simulate if it is possible for Gabriel to win with that particular X-omino. If we can find any X-omino that Richard can pick which results in Gabriel being unable to win, then we report that Richard can win. But if Gabriel wins for all X-ominoes that Richard picks, then we report Gabriel as the winner.

Simulating Richard picking a X-omino is straightforward. Richard can pick each of the X-omino one by one. The trickier part is to check if Gabriel can win, we now describe a strategy to

perform this check.

We take the RxC grid and we have the X-omino that Richard has required that Gabriel use at least once. We can brute force the placement of this X-omino in the RxC grid (NOTE: We have to try the CxR grid too, we elaborate on the reasons to take the CxR grid in a subsequent paragraph). If it is impossible to place the X-omino (e.g. if the width of the X-omino is bigger than C) in either the RxC grid or the CxR grid then we trivially say that Richard wins for that particular X-omino. If it is possible to place the X-omino in a particular location, we still need to check whether it is possible for Gabriel to win. Let's see some examples. Suppose that the X-omino Richard chose is the following.

```
. ##
## .
```

and we are given a 2x4 grid. In that case, we can place the X-omino in the following two configurations.

```
. ## .
## . .
```

or

```
. . ##
. ## .
```

We notice that in both configurations, the X-omino has divided the grid into two connected components of size 1 and 3. Well, we had mentioned earlier in the section "Available cells is a multiple of X" that such configurations imply Gabriel will always lose.

In fact, after placing a X-omino if it results in connected components of size M where all such M is a multiple of X, then we can say that Gabriel wins with that X-omino placed at that particular location.

As another example, if we use the following 4-omino:

```
##
##
```

and we are given a 2x6 grid, and if we placed the 4-omino as follows:

```
. ## . . .
. ## . . .
```

then it is not possible for Gabriel to win (there are two connected regions of size 2 and 6, and 2 and 6 are not multiples of 4). But if we placed the 4-omino as follows:

```
## . . . .
## . . . .
```

or

```
. . ## . .
. . ## . .
```

Then in both configurations we say that Gabriel wins (in the first case, there is one connected component of size 8, and 8 is multiple of 4; in the second case, there are two connected components of size 4 each, and 4 is a multiple of 4).

As mentioned in an earlier paragraph, we try the CxR grid too. The reason is because we have to check for a 90 degree rotation of the X-omino that Richard selected. Remember, Gabriel can rotate or reflect the X-omino when initially placing it on the grid. Instead of rotating the X-omino, we can instead just rotate the grid! Therefore it suffices to check if we can fill a RxC grid, or a CxR grid with that X-omino. Note that we don't need to consider all possible reflections and rotations of the X-omino because in grid space, things are symmetric therefore it suffices to check for both the RxC and CxR grid.

Therefore to generalize it, after placing the selected X-omino at a particular place, we can check the sizes of the edge-connected '.' components, and if all such components have size M is a multiple of X, then it means that we can always fill the M space with M/X X-ominoes. In such a case, we say that that particular grid configuration is one for which Gabriel can win. An explanation of this winning condition for our brute force solution is based off the conclusion that we arrive at from the "Alternative solution" presented below.

Alternative solution

The alternative solution involves careful analysis of various cases. Let $S = \min(R, C)$ and $L = \max(R, C)$ so that $S \leq L$. Suppose that the grid dimensions are $S \times L$ (if the dimensions are $L \times S$, the grid can be rotated without affecting the win conditions).

Richard can force a win if any of the following conditions hold; otherwise Gabriel will win.

1. X does not divide $S \cdot L$,
2. $X=3$ and $S=1$,
3. $X=4$ and $S \leq 2$,
4. $X=5$ and either (i) $S \leq 2$ or (ii) $(S, L) = (3, 5)$,
5. $X=6$ and $S \leq 3$,
6. $X \geq 7$.

We have already explained (1) and (6) in the previous paragraphs. For (2), (3), (4i), (4ii) and (5), Richard can choose the following pieces and always guarantee a win:

(2)

```
# .
##
```

It is impossible for Gabriel to fit the above piece in a grid with $S=1$. Similar explanation follows for the following two cases.

(3)

```
###
.#.
```

For the above piece, when the grid has $S=2$ notice that it will divide the '.' cells into two connected regions and that it is impossible for these regions to have a size which is a multiple of 4. It is impossible for Gabriel to fit the above piece in a grid with $S=1$.

(4i)

```
#..
##.
.##
```

Similar to the above cases, it is impossible for Gabriel to fit the above piece in a grid with $S \leq 2$.

(4ii) For this case, Richard can use the same piece as used in (4i). By trying all possibilities, one can see that it is impossible for Gabriel to win with $X=5$ and a 3×5 grid.

```
#....  .#...  ..#..
##...  .##..  ..##.
.##..  ..##.  ...##
```

(5)

```
.#..
####
.#..
```

A similar explanation follows for the above piece as the explanation for (3).

For all combinations of X , S and L not satisfied by the above condition, Gabriel will win. We provide here an explanation for the $X=6$ case, and leave the other cases as an exercise. Let's consider the 4×6 grid, which is the smallest grid for which $S > 3$ and $S \cdot L$ is a multiple of X . We show below Gabriel's strategy for filling the 4×6 grid, and then generalize for cases where the grid is bigger than 4×6 .

First off, we point out that for a 6-omino, there are only [35](http://en.wikipedia.org/wiki/Hexomino) choices. In fact, listed below are the number of choices for each X -omino:

- 1-omino, 1 choice,
- 2-omino, 1 choice,
- 3-omino, 2 choices,
- 4-omino, [5 choices](#),
- 5-omino, [12 choices](#).

Since the 6-omino only has 35 choices, we list out below a case-by-case analysis of the 35 choices that Richard can make, and Gabriel's strategy for filling up the rest of the cells to guarantee a win for himself. '#' denotes the original piece that Richard chooses while 'a', 'b', and 'c' are 6-ominoes that Gabriel chooses.

#####	a#####	a#####	a#####	####cc	##abbb	##abbb
aaaaaa	aaaaa#	aaaa#c	abb#cc	aab##c	##abbb	#aabbb
bbbbbb	bbbbbb	abbbbc	aabbcc	aabbcc	#aacc	#aacc
ccccc	ccccc	bbcccc	aabbcc	aabbbc	#aacc	#aacc
##abbb	#aabbb	###bbb	#aaaa	#aaaa	aaa#bb	aa#bbb
#aabbb	##abbb	#aabbb	###cca	####ca	a####b	a####b
#aacc	##aacc	#aacc	#cccb	#ccccc	aacc#b	aaac#b
##aacc	#aacc	#aacc	#bbbb	bbbbbb	cccbb	ccccb
a#bbbb	aa#bbb	aa#bbb	a###bb	###bbb	###bbb	###bbb
a####b	a####b	a####b	aaa#b	#a#bb	aa###b	a###bb
aacc#b	aaa#bb	aa#ccb	acc#bb	aaaccb	aaccb	aaaccb
aaccc	ccccc	acccb	acccb	aaccc	aaccb	aaccc
###aaa	aaa#bb	aaa###	aa#bbb	###bbb	###bbb	aaaaaa
###aaa	a###bb	aaab##	a###bb	aa#bbb	aa#bb	###bbb
ccbbb	aac#b	bbbb#c	aaa#b	aa#cc	aaa#cb	#c#bbb
ccbbb	ccccc	bccccc	ccccc	aaccc	acccc	#ccccc
aaaaab	aabbbb	#bbbbb	aaaaaa	aaaaab	aaaabb	##aaaa
#a#cbb	a#b#bc	a###bb	#bbbbc	a#bbb	aa#cbb	c#aab

###cbb	a###cc	aaa#cc	##bbcc	###ccb	###cbb	cc##bb
#ccccb	aa#ccc	aacccc	###ccc	#cccc	##cccc	cccbbb

Whew, that was a lot of cases! Clearly, Gabriel will always win with a 4x6 grid and any 6-omino that Richard chooses.

We now explore the case when the grid is bigger for $X=6$. Let's pick the 6x8 grid. To win in such cases, Gabriel can use the following strategy. Gabriel can use the top-left corner of the board to place the piece that Richard chooses so that he can complete a 4x6 portion similar to the manner described in the 35 cases above. Then Gabriel can label the remaining cells using a [Hamiltonian path](#) (with cells as vertices, and adjacent cells as neighbors). 'Z' denotes the 4x6 portion on the top-left corner of the grid, and the rest is filled with a sequence of 'a'-'x' which denotes the Hamiltonian path. In this case, Gabriel can start the path from the top-right corner and 'snake' back and forth. In general, depending on the parity of the number of rows of the 'Z' region (let's call the number of these rows T), Gabriel can start from the top-right (when T is even), or from the cell adjacent to the top-right corner of the Z-region (when T is odd) then 'snake' back and forth to fill the rest of the cells.

```

ZZZZZZba
ZZZZZZcd
ZZZZZZfe
ZZZZZZgh
ponmlkji
qrstuvwxyz

```

Observe that the cells 'a'-'x' forms a chain. We can simply chop off this chain into sizes of 6 each (i.e. a 6-omino!). This way, Gabriel can win for $X=6$ and grids that are bigger than 4x6.

Similar to above, we can list out cases for the other X-ominoes, which we leave as an exercise.

Here is a sample implementation in Python:

```

def richard_wins(X, R, C):
    S = min(R, C)
    L = max(R, C)
    if (S * L) % X != 0: return True
    if X == 3 and S == 1: return True
    if X == 4 and S <= 2: return True
    if X == 5 and (S <= 2 or (S, L) == (3, 5)): return True
    if X == 6 and S <= 3: return True
    if X >= 7: return True
    return False

for tc in range(input()):
    X, R, C = map(int, raw_input().split())
    print "Case #%d: %s" % (tc + 1,
        "RICHARD" if richard_wins(X, R, C) else "GABRIEL")

```