`.img` of the Artist as Filipino

Actions are Objects

Let's begin with a seemingly-unrelated and pointlessly-abstract observation. What is the number 2? What is the number 3? And what does $2+3$ mean?

One person might answer that they are *objects*, like nouns.

- I have 2 coins.
- I have 3 coins.
- $2+3=5$, because if I combine my piles of 2 and 3 coins, I have a pile of 5 coins.

But another person might answer that they are *actions*, like verbs.

- I will give you 2 coins.
- I will give you 3 coins.
- $2+3=5$, because if I give you 2 coins and then give you 3 coins, then that is equivalent to if I had just given you 5 coins.

Interestingly, both are valid! In modern mathematics, we use numbers to refer to both objects *and* actions. The more prescient insight is that *we can treat actions like objects*. Much like numbers or ingredients in a recipe, we can take two actions and "combine" them together in order to get another action.

Functions are Objects

Recall that a **function** is a mathematical object that takes some input, performs some calculations with it, and then returns some output. It would be valid for us to say that the function **transforms** the input into some output, so we will somewhat interchangeably also use the word "transformation" to refer to functions.

For example, I can define a function that takes in a square grid as input, and its output is what the resulting grid would be if I rotated it 90° clockwise.

```
      ---
1---2  |          4---1
|   |  V          |   |
|   |    ---->    |   |
|   |             |   |
4---3             3---2
```

Thus, a function can be seen as an *action* performed on the input. However, recall that we said that *functions are mathematical objects*, so we can treat them like objects. We can define an operation that takes two functions and combines them into another function.

For example, we can think of the "and then" operation. If f and g are functions, then we can chain them into one composite function: consider the transformation we get if we first apply f *and then* apply g.

For instance, suppose we are again working with square grids. Let f be the transformation, "Rotate 90° clockwise" and let g be the transformation, "Reflect across the vertical axis". We can chain them together into the composite function, "Rotate 90° clockwise *and then* reflect across the vertical axis".

```
    ---             .
1---2 |         4---1           1---4
|   | V         | . |           |   |
|   |   ---->   | . |   ---->   |   |
|   |           | . |           |   |
4---3           3---2           2---3
                  .
```

But we can think of a simpler name for this composite function, no? By inspection, we see that "Rotate 90° clockwise *and then* reflect across the vertical axis" is equivalent to if we had just performed the action, "Reflect across the main diagonal."

```
\
 1---2           1---4
 |\  |           |   |
 | \ |   ---->   |   |
 |  \|           |   |
 4---3           2---3
      \
```

The `and then` operation might be more familiar to you under the name *function composition*, although you have to be careful with the order---because of modern function notation, $g(f(x))$ is equivalent to `f and then g` applied to `x` (it's reversed because modern functions with nested arguments are evaluated inside-out).

Rotations and Reflections on a Square

In the problem, we are given four types of transformations in the input:

- `L` Rotate 90° counterclockwise
- `R` Rotate 90° clockwise
- `H` Reflect horizontally (across the vertical axis)
- `V` Reflect vertically (across the horizontal axis)

What we can do is pull out some pen and paper and record what happens when we combine all possible pairs of transformations using the *and then* chain operation. Literally, I encourage you to pick up a labeled square and then try flipping it around in your hands and seeing what happens.

Here are some highlights:

- If we apply `L` and then `R`, this is the same as doing nothing.
- If we apply `H` and then `H`, this is the same as doing nothing.
- If we apply `L` and then `L`, this is the same as rotating 180°.
- If we apply `H` and then `V`, this is the same as rotating 180°.
- If we apply `R` and then `H`, this is the same as reflecting across the main diagonal.
- If we apply `H` and then `R`, this is the same as reflecting across the anti-diagonal.

Note that the last two bullets show that **order matters** with the "and then" operation.

Let's encode these new transformations that we've discovered using the following symbols:

- `0` Do nothing
- `U` Rotate 180°
- `D` Reflect across the main diagonal
- `A` Reflect across the anti-diagonal

Let's create a "multiplication table" that summarizes all possible results from combining our original four operations. Let the row indicate which transformation is done first, and let the column indicate which transformation is done second.

	L	R	H	V
L	U	0	A	D
R	0	U	D	A
H	D	A	0	U
V	A	D	U	0

Actually, what if we made a more complete multiplication table that also includes the newly discovered transformations?

	0	L	U	R	D	V	A	H
0	0	L	U	R	D	V	A	H
L	L	U	R	0	H	D	V	A
U	U	R	0	L	A	H	D	V
R	R	0	L	U	V	A	H	D
D	D	V	A	H	0	L	U	R
V	V	A	H	D	R	0	L	U
A	A	H	D	V	U	R	0	L
H	H	D	V	A	L	U	R	0

I'm sure you're already able to see some patterns in the table (ordering the transformations in this way was an intentional choice on my part).

It turns out that there's something nice about this set of transformations `{0, L, U, R, D, V, A, H}`:

- If I combine any two transformations from that set using the "and then" operation, I get another element of that set. With fancy jargon, this set is **closed** under the "and then" operation.
- There is a "do nothing" transformation represented by `0`, such that `0 and then x` is the same as `x and then 0` which is the same as `x`.
- Each transformation has an "inverse" such that applying it *and then* its inverse is equivalent to the "do nothing" transformation `0`.

Additionally, there is one more crucial property that you can verify, which I call "parentheses don't matter".

- Let `x` and `y` and `z` be transformations. Then, `(x and then y) and then z` gives the **same result** as `x and then (y and then z)`.
- Thus, it makes sense for us to drop the parentheses and just say `x and then y and then z`.
- Note that the order of the transformations still has be preserved though.

You can try substituting in different transformations into `x`, `y`, and `z`, and you will see that it holds true. You might be aware that this property has the fancy name of **associativity**, and it is an inherent property of the `and then` operation.

(Subtask 1) Implementation

What does this mean for our problem?

For convenience, let `xy` be understood to mean "Apply `x` and then apply `y`" (i.e. the `and then` is implicit and can be omitted).

- **Any** command string can be "simplified" to one of eight possible transformations by repeatedly "reducing" the expression, just like we would with arithmetic.
  - For example, `LDRV = LDA = LU = R` (you can check this).
- Therefore, the answer to any `?` query will **always** be only one of `8` possibilities.
- Thus, *only once at the start*, we can apply each transformation to the given mosaic, and then store the resulting hash value as a result of this transformation.
- Answering each `?` query then reduces to finding which transformation it corresponds to, and then lookup up its corresponding precomputed hash value.

We can also use our multiplication table to make the *implementation* of all these transformations easier! Let `orig` be the original grid and let `res` be the post-transformation grid.

- `D` is simple to implement: we just have `res[i][j] = orig[j][i]`.
- `H` is simple to implement: we just have `res[i][j] = orig[i][n-1-i]` (assuming 0 indexing).

But note that **all other transformations can be expressed in terms of `D` and `H`!**

- `0 = DD` (in practice, though, `0` is trivial to implement)
- `L = HD`
- `U = HDHD`
- `R = DH`

- D = D
- V = DHD
- A = HDH
- H = H

So we only need to implement D and H, and all other transformations can be obtained by chaining H and D together with and then (function composition). With fancy jargon, we say that H and D **generate** {0, L, U, R, D, V, A, H}.

(Subtasks 2 and 3) Standard Data Structures which play nicely with "and then"

How again do we use prefix sum tables in order to compute a[L] + a[L+1] + ... + a[R]?

- Precompute all the prefixes P[i] = a[1] + a[2] + ... + a[i]
- Thus, a[L] + a[L+1] + ... + a[R] = -P[L-1] + P[R]

But think about it---what properties of addition did we even end up using here?

- Every number x has an additive inverse $-x$
- (Implicitly) Addition is associative, so we can ignore parentheses.

But hey! Our set {0, L, U, R, D, V, A, H} under the and then operation **also** has inverses and is associative!

- Precompute all the prefixes P[i] = s[1] and then s[2] and then ... and then s[i]
- Thus, s[L] and then s[L+1] and then ... and then s[R] = (P[L-1] inverse) and then P[R].

Prefix "sums" allow us to solve subtask 2, where there are no updates to the reference string, in $\mathcal{O}(1)$ per question.

Similarly, if you look at the implementation of segment trees, you'll note that addition is not at all that important to its core details (which is in fact why you might encounter problems which ask for range min or range XOR queries on segment trees).

If you implement a segment tree using the and then operation on these transformations, you can solve subtask 3 in $\mathcal{O}(\lg n)$ per question and update.

(100 pts) Permutations are transformations too

We might have an inkling that we need to create a lazy segment tree of some kind in order to process range updates. But how?

Subtask 5 gives us a hint. Consider the simpler problem---what if the relabeling is always done to entire reference string?

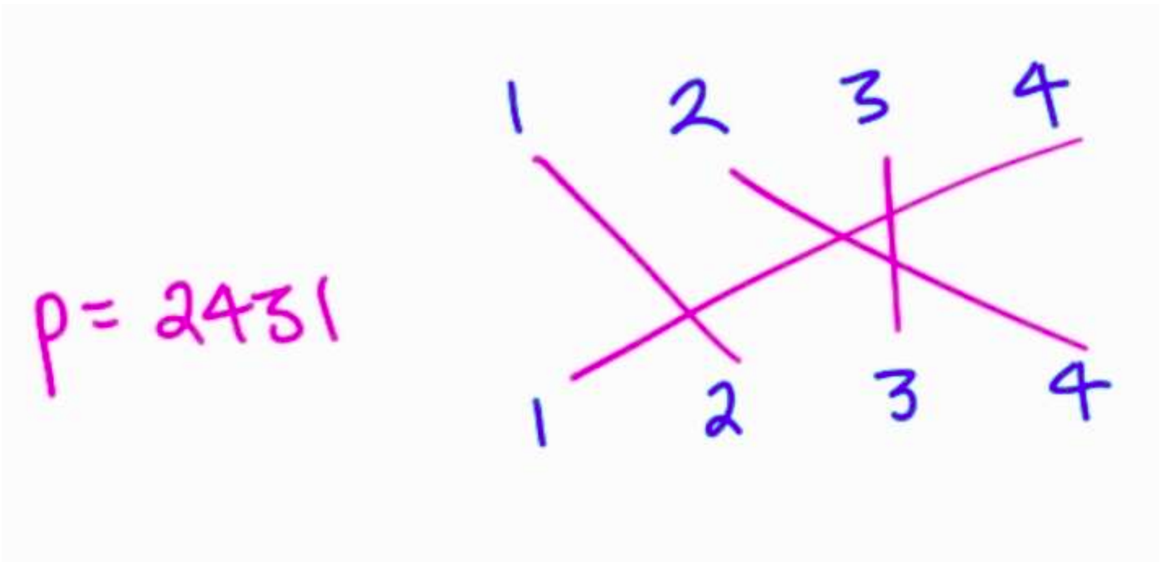
The key is to note that the p in each update query is always a permutation. From combinatorics, you might be familiar with permutations as *objects*, but actually, you can also think of permutations as *actions*.

Let $p = p_1p_2p_3p_4$ be a permutation of the integers from 1 to 4. Then, we can define a function φ such that $\varphi : k \mapsto p_k$ (or you might prefer the notation $\varphi(k) = p_k$). In words, it "sends" each k to p_k.

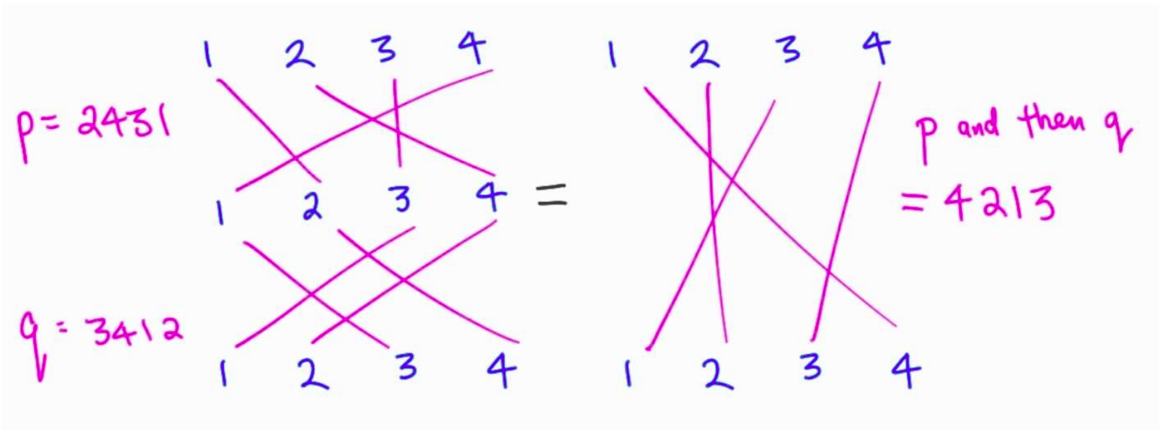
For example, `p = 2431` induces a function φ where

- $\varphi(1) = 2$
- $\varphi(2) = 4$
- $\varphi(3) = 3$
- $\varphi(4) = 1$

It's common to visualize these by drawing the input in a row on top, and the output in a row on the bottom, and then connecting each input to its output.



And hey! These are functions! So we can apply the `and then` operator to combine two permutations together. For example, if `p = 2431` and `q = 3412`, then `p and then q = 4213`, which is best visualized in the following diagram.



Here are some facts that you will want to verify for yourself:

- Combining any two permutations with the `and then` operation gets you **another** permutation. So, using the jargon, the set of permutations is **closed** under the operation `and then`.
- There is a permutation which corresponds to the "do nothing" transformation
- Every permutation has a corresponding "inverse"

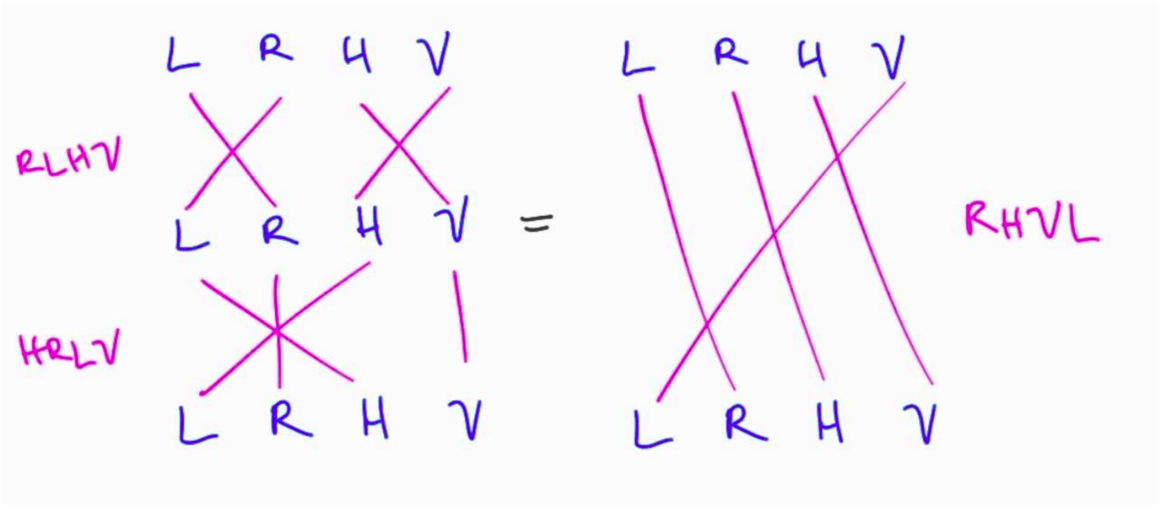
Only the first fact is important for our solution, but the other two are also worth verifying for yourself because it shows that permutations have a similar structure to our grid transformations from earlier!

This leads to a solution for Subtask 5. Instead of permutations on 1 to 4, consider permutations of `LRHV` (but our discoveries from earlier still apply because this is completely equivalent).

- Create **24 different reference strings**, one for each of the $4!$ permutations of 4 elements (where the original reference string is relabeled using each permutation).
- From these, create **24 different prefix "sum" tables**
- Let the variable `curr` correspond to the "current" permutation of `LRHV` (initialized to the `do nothing` permutation)
- Whenever an update `! 1 m p` query appears, update the current permutation by applying `curr = curr and then p`.
- Whenever a `?` query appears, consult the prefix sum table of the permutation that corresponds to the current relabeling of `LRHV`.

For example, if the reference string is `LLRHVR`, then...

- The current permutation is `LRHV` (do nothing)
- After receiving `! 1 6 RLVH`, the current permutation becomes `LRHV and then RLVH = RLVH`. So, for any questions, we consult the prefix sum table of the reference string `RRLVHL`
- After then receiving `! 1 6 HRLV`, the current permutation becomes `RLVH and then HRLV = RHVL`. So, for any questions, we consult the prefix sum table of the reference string `RRHVLH`.
- ...and so on



This allows us to solve subtask 5 in $\mathcal{O}(24)$ per update and $\mathcal{O}(1)$ per question.

To get full points, we apply this same insight to a segment tree with lazy propagation.

- Let **each node in the segment tree** have 24 versions, one for each permutation.
- Let each node's lazy flag be in the form of a `curr` permutation, for that node's current permutation.
- If you were to lazily "push" an update onto the entire interval represented by a node, we only need to update its lazy flag using the `and then` operation.
- Whenever you "pull" (i.e. recompute the value of some node), you reset its `curr` to

the `do nothing` permutation and you also have to recompute each of the 24 versions of itself.
- Otherwise, treat it like a normal lazy segment tree.

This solution runs in $\mathcal{O}(24 \lg n)$ per query of either kind.

In order to fit it under the time limit, you may also need to do some or all of the following optimizations:

- Represent permutations as integers from 0 to 23 instead of literally as a list of 4 elements.
- Precompute the 24×24 "multiplication table" for the permutations instead of computing it ad hoc.
- Be careful to only pull (recompute) a node when it is necessary, since have to do it 24 times.

Group Theory

We've demonstrated how addition of integers, rotations and reflections of a square, and permutations of a group of objects *all* share some common features:

- They're associative
- They have "do nothing" elements
- They have inverses

These are all examples of what mathematicians call **groups**. Groups are usually the introductory topic of the wide field of math known as Abstract Algebra. They're incredibly interesting, and I encourage you to study them on your own time! I offer two main arguments for why they're so interesting.

First is a pragmatic argument. As we saw with prefix sum tables and segment trees, we can create algorithms that are more *general*. We identified that some, but not all, features of the integers were key to making those ideas work. For example, addition of integers is commutative, but we recognized that the same technique could still be applied even to objects that aren't. By being as *abstract* as possible, distilling the objects needed to their very *essence*, we are able to create ideas that are more useful due to being more widely applicable to a variety of different objects. If you stick to computer science, you'll recognize these ideas when you start implementing template classes, interfaces and abstract classes, and (if you dare) the entire rich field of Type Theory.

Second is an aesthetic argument. By assuming *only those three axioms* about a set and an operation, we are able to create a robust theory of mathematics that's fit for a class spanning an entire semester (and possible more). Isn't it interesting how so many rich and powerful results can be derived from *just three simple rules*? These are the simplest structures needed for us to meaningfully do algebra, and all the same, we actually have so much that we can say about them! If you stick with mathematics and computer science, then you will see more structures like rings, fields, and even automata and Turing machines, whose beating heart is the stunning amount of complexity that can be reaped from a few simple rules.

I hope this problem got to concretely demonstrate some of these arguments to you, and I hope you feel enticed to learn more about groups and abstract algebra!