NOI 2018 Editorial

Mysterious Array Solution

We are given Q constriants of the form "the minimum value between positions a and b is x".

Count the number of permutations of the numbers $1, 2, \ldots, N$ that satisfy all constraints.

Initial Insights: $N, Q \leq 10$

We have to first understand what each constraint actually implies.

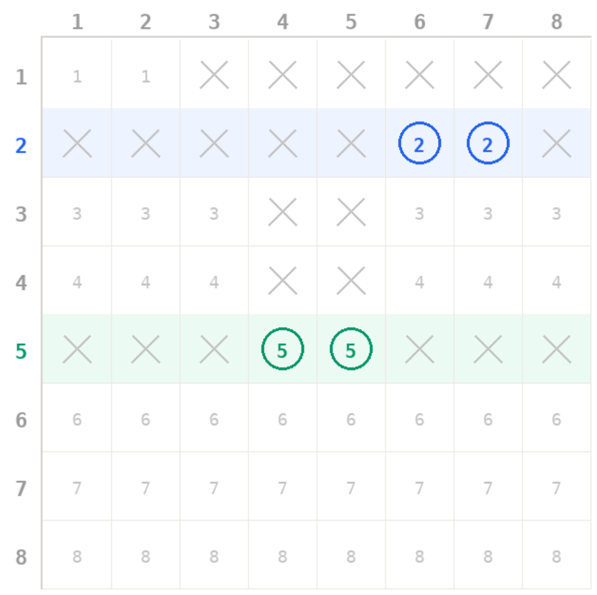
For each constraint (a, b, x), we know get two important pieces information:

- The element x must be in the range $[a, b]$.
- No elements $< x$ can be in the range $[a, b]$.

Consider the sample input 2:

```
8 3
3 7 2
6 8 2
4 5 5
```

We can draw this out in a grid, each column showing what elements could be placed there.



Using this, we can create brute-force solutions.

One idea is to try all 10! permutations, and iterate through all constraints ($O(Q)$), and verify if constraints are satisfied ($O(N)$) by looping through the interval $[a, b]$. However, this might be too slow since it runs in $O(N! \cdot NQ)$. One could remove a factor Q by precomputing the grid above, and for each permutation check if each element in the permutation is valid by checking with the grid.

Another idea is to write a DP with 2^N states, a bitmask DP. Each state is represented by a bitmask of size N, where each bit represents if the corresponding position has been occupied. The number of active bits will indicate which element is being considered to be placed down next. The transition of the DP would be to try putting down the next element at all possible $O(N)$ positions.

```python
def dp(mask):
    i = popcount(mask) + 1 # next value to place
    if i > N:
        return 1
    ans = 0
    for j in range(1, N + 1):    # try each position
        if mask & (1 << j):      # position j already taken
            continue
        if not valid[i][j]:      # precomputed grid
            continue
        ans += dp(mask | (1 << j))
    return ans
```

This gives us a $O(N2^N)$ solution.

Erm, what now? $N, Q \leq 1000$

Since we want to calculate some combinatorial structure given some constraints, we need some nice way of ordering the things we are calculating so that we can easily count them.

Some common ways to think in this case would be,

- At each position, what numbers can be placed here?
- For each number, what positions can it placed at?

We will go with the second idea, since we will soon realize it works really well in this situation.

Consider if the constraints only consisted of the type

- No elements $< x$ can be in the range $[a, b]$.

If two or more constraints cover the same interval, only the largest x matters.

We can then imagine that we are placing down the numbers one-by-one, from 1 to N. This is easy to count, since all positions that 1 can be placed, the number 2 can also be placed. Thus, if we have counted the number of positions where each number i can be placed at is c_i, then the answer is simply

$$\prod_{i=1}^{N}(c_i - (i - 1))$$

since when we place number i, exactly $i - 1$ positions have already been taken by smaller numbers, and all of those positions are a subset of the positions available to i.

But our constraints also have the second type:

- The element x must be in the range $[a, b]$.

This complicates things, because now certain numbers are *forced* into specific positions. We need to account for this carefully.

Consider a constrained value x with range $[a, b]$. Since x is the minimum of $[a, b]$, every position in $[a, b]$ must hold a value $\geq x$. If multiple constrained values cover the same position, only the largest one imposes the tightest restriction. We call this largest constrained value the **dominator** of that position.

For an $O(N^2)$ approach, we can compute this directly: for each position j, loop through all constrained values and find the largest one whose range covers j.

```
dom = [0] * (N + 1)
for j in range(1, N + 1):
    for each constrained value x with range [a, b]:
        if a <= j <= b:
            dom[j] = max(dom[j], x)
```

Since each position has exactly one dominator (assuming that $a = 1$, $b = N$, and, $x = 1$), the positions partition into disjoint groups. One group per constrained value.

A constrained value x that appears in multiple constraints $(a_1, b_1, x), (a_2, b_2, x), \ldots$ must lie in the intersection $[\max(a_i), \min(b_i)]$. But it can only be placed at a position it *dominates*, otherwise some larger constrained value $y > x$ also needs that position to hold a value $\geq y$, and placing x there would violate that.

So the positions dominated by x split into two kinds:

- Positions inside x's mandatory intersection, value x **can** be placed here.
- Positions outside the mandatory intersection, value x **cannot** go here, but larger unconstrained values can.

We place values from 1 to N in order, maintaining a pool of available positions (initially empty):

- If value i is **constrained**: all positions dominated by i enter the pool. Multiply the answer by the number of those positions that lie inside i's mandatory intersection (the valid spots for i). Then decrement the pool by 1 (value i takes one spot).
- If value i is **unconstrained**: it can go anywhere in the pool, since all pool positions require a value $\geq$ some smaller number. Multiply by the current pool size, then decrement by 1.

Why is there no interference? Positions dominated by constrained value x only enter the pool when we reach $i = x$. Since we process values in increasing order, no earlier value could have taken them. And unconstrained values are always large enough to satisfy whatever constraint dominates their chosen position.

The dominator computation is $O(NQ)$ and the counting step is $O(N)$, giving $O(NQ)$ overall.

Speeding up: $N, Q \leq 2e5$

The bottleneck in the previous solution is computing the dominator of each position.

Notice that as we move from position j to $j+1$, the set of constrained values covering the position only changes slightly some values start covering us, and some stop. We can solve it with a sweep line.

For each constrained value x with range $[a, b]$, we create two events:

- At position a: insert x into the active set.
- At position $b+1$: remove x from the active set.

Note that a value x may appear in multiple constraints, say (a_1, b_1, x) and (a_2, b_2, x). We create events for each constraint separately, so x may appear in the active set multiple times. As we sweep from position 1 to N, we maintain a multiset or priority queue of active constrained values. At each position, the dominator is simply the maximum element in the multiset, which a `multiset` supports in $O(\log Q)$ time.

```
for each constraint (a, b, x):
    events[a].append((+1, x))
    events[b + 1].append((-1, x))

active = multiset() # or priority queue,
for j in range(1, N + 1):
    for (type, x) in events[j]:
        if type == +1:
            active.insert(x)
        else:
            active.remove(x) # remove one copy
            # if we use an priority queue, we can keep track of removed elements and
                remove it if they are at the top

    dom[j] = the largest element in active
```

The total time for the sweep is $O((N+Q)\log Q)$.

Combined with the $O(N)$ counting step from the previous section, the overall complexity is $O((N+Q)\log Q)$.