Mán lì (manli)

Author: Alexandru Gheorghies

Developer: Alexandru Gheorghies

Solution

This problem asks the following:

- Given an undirected graph, find the lexicographically minimum coloring of its complement.

This problem can be solved with the following greedy algorithm:

- Initially, set $\text{col}[u] = 0$ for all $1 \leq u \leq n$.
- Then, for every node u from 1 to n, set its color $\text{col}[u]$ to $\text{MEX}(0, \text{col}[v_1], \text{col}[v_2], \ldots, \text{col}[v_k])$, where $[v_1, v_2, \ldots, v_k]$ is the list of all nodes adjacent to u in the complement graph.

  Since, at this point, $\text{col}[v_i] = 0$ for all $v_i > u$, these nodes can be ignored when computing the color of node u.

This greedy algorithm can be implemented in $O(N + M)$ in the following way:

- We will maintain a frequency array `fr` of the colors that have already been computed. Initially, as we have not computed the color of any node, $\text{fr}[c] = 0$ for every color $1 \leq c \leq N$.
- Additionally, we will maintain a variable global_mex $= \text{MEX}(0, col[1], col[2], \ldots, col[n])$. Initially, global_mex $= 1$.
- For every node u from 1 to n:
  1. We will initialize $\text{col}[u]$ with the value of global_mex.
  2. For every node $v < u$ adjacent to u in the input graph, $\text{fr}[\text{col}[v]] := \text{fr}[\text{col}[v]] - 1$. If $\text{fr}[\text{col}[v]] = 0$, then $\text{col}[u] := \min(\text{col}[u], \text{col}[v])$.
  3. Now we have computed $\text{col}[u]$, and, as such, we will have to update fr. For every node $v < u$ adjacent to u in the input graph, $\text{fr}[\text{col}[v]] := \text{fr}[\text{col}[v]] + 1$ (undoing step 2).
  4. Then, $\text{fr}[\text{col}[u]] := \text{fr}[\text{col}[u]] + 1$.
  5. Finally, while $\text{fr}[\text{global\_mex}] > 0$, do global_mex := global_mex + 1.

Since global_mex will be updated at most N times (step 5) and fr will be updated $M + M + N$ times (steps 2, 3 and 4), the total time complexity is $O(N + M)$.

Mìmì (mimi)

Author: Cristian Luchian

Developer: Bogdan-Ioan Popa,Cristian Luchian

Solution

Our solution finds the answer with an average number of queries equal to $n/3 + 2$. To do this we execute multiple `query_cell` operations in such a way that no two queries have the same row or the same column (this is to make sure we cover as much space as possible). After an average of $n/3$ queries we will either find an element that is $< n$ or that can be written in the form $k * n$ where k is an integer. Given the way the matrix is shuffled this means that we will either get the row or the column of the element 0. Now we can ask what is the opposite coordinate (meaning we ask where is 1 if we already know where a number with the form $k * n$ or we ask where is n if we already know where is a number $< n$). After that we will know both coordinates a and b of 0, but given that the matrix might be rotated we cannot be sure which one is the row and which one is the column. As such, we can do one more `query_cell` to find out which of the cells (a, b) or (b, a) contains the value 0.

But why does it work? You can imagine it as having two concatenated rows and each of them has a marked element. For simplicity you can assume that you have an array with $2 * n$ elements out of which exactly 2 are marked and you want to find one of them by asking random elements. The exepected number of elements you have to query is $2 * n/3$ but since you can query 2 elements per step the average becomes $n/3$.

Shù Yìn (shuyin)

Author: Alin-Gabriel Raileanu

Developer: Alin-Gabriel Raileanu

Solution

Key observation: The yìn uniquely determines the unlabeled structure of the tree — in particular, for each node v, it determines its number of children c_v. What remains is to count the number of ways to assign labels $1, 2, \ldots, N$ to the nodes such that the DFS visits them in the order induced by their labels.

Formula: Since the root is always labeled 1, we need to assign the remaining $N-1$ labels to the $N-1$ non-root nodes. Consider all $(N-1)!$ ways to do so. For a given assignment to produce the correct yìn, the DFS must visit the children of every node v in increasing order of their labels. Among all $c_v!$ orderings of the children of v, exactly 1 satisfies this condition. Since the choices for different nodes are independent, we divide the total by $c_v!$ for each node v.

This is a **permutation with repetition**: we arrange $N-1$ distinct labels, but the c_v children of each node v are interchangeable in the sense that only one of their $c_v!$ relative orderings is valid. The answer is therefore:

$$\text{answer} = \frac{(N-1)!}{\prod_{v=1}^{N} c_v!}$$

Time complexity: $O(N)$.

Space complexity: $O(N)$.

Yánshí (yánshí)

Author: Bogdan-Ioan Popa, Daniel Gheorghe

Developer: Bogdan-Ioan Popa, Daniel Gheorghe

Solution

We count valid subarrays by fixing the right endpoint r and asking how many left endpoints $l \leq r$ make $A[l \dots r]$ *suntzu* (every value present in the subarray appears K times). As r moves from left to right we maintain, for every candidate left l, a numeric score that equals the number of positions which contain a value not currently satisfied (appear > 0 but $< K$) in $A[l \dots r]$. A left l is valid exactly when its score is 0.

We implement that score with a difference-style segment tree: leaves correspond to left indices l, and point updates at certain occurrence positions are used to apply the effect of adding the new element $A[r]$ to all $l \leq r$ ($+1$ on prefix $[1 \dots r]$). Let x be some value and $pos_1, pos_2, ..., pos_T$ be the positions where x occurs, from right to left, starting from r. Then we will do a $-K$ on segment $[1 \dots pos_K]$ and then -1 on every segment $[1 \dots pos_{K+1}], [1 \dots pos_{K+2}]$, etc. The segment tree must report the minimum value on the prefix and its frequency.

Each r does O(1) point updates and one query on the segtree, so the solution runs in $O(NlogN)$ time.

Píngjun shù wèntí (pingjun)

Author: Robert-Sebastian Moldovan

Developer: Robert-Sebastian Moldovan

Solution

A brute force solution in $O(N * Q)$ would involve running a shortest path algorithm from node K to any node in $[L, R]$. Since the graph is a tree and we have a unique path to all other nodes, we can use a BFS or a DFS instead of Dijkstra.

Another brute force solution in $O(N * Q)$ would involve precomputing a sparse table for $O(1)$ LCA computation, and computing node-node distances in $O(1)$.

Both of these solutions should score around 15 to 27 points, depending on how efficient the contestant's implementation is.

A suboptimal solution involves partitioning the nodes into buckets of size B and precomputing multiple-source shortest paths from them. This can be done with either Dijkstra's algorithm or tree DP in $O(N * B * log(N))$ or $O(N * B)$ respectively. When we have a query $K, [L, R]$, we consider all precomputed buckets of the form $[(P-1)B+1, PB] \subseteq [L, R], P \in \mathbb{N}$, and brute forcing through all other nodes using the sparse table mentioned earlier for LCA.

Depending on whether Dijkstra or tree DP is used, this solution can score between 60 to 75 points.

A better solution involves using centroid decomposition to build a centroid decomposition tree, which we can use to represent paths a lot more efficiently. Namely, we can represent all $O(N^2)$ distinct paths in the tree as a concatenation of two paths of the form $u, par(u)$, $u, par(par(u))$, ..., which there are $O(N * log(N))$ of in total, where $par(u)$ denotes u's parent in the centroid decomposition tree. We can build this tree by finding the centroid of a given subtree, splitting that subtree up into separate subtrees disconnected from the centroid, and repeating that process on all of those subtrees recursively, drawing edges between "adjacent" centroids in this process.

Having built the centroid decomposition tree, we can now add all nodes to themselves and all of their ancestors' sets in the centroid decomposition tree, with pairs of the type $(dist(u, v), u), v \in Ancestors(u)$. Having done this, to answer a query $K, [L, R]$, it is enough to take the minimal pair of the form $(dist(K, v) + dist(u, v), u)$, where $L \leq u \leq R$ and $v \in Ancestors(K)$.

Depending on how the set is implemented and the overall constant factor of the implementation, this solution can score anywhere between 60 to 100 points. The best solution involves storing these pairs in sparse tables and binary searching the ranges we need to query, thus obtaining $O(N * log^2(N))$ space and time complexity.

Sàima chang (saimachang)

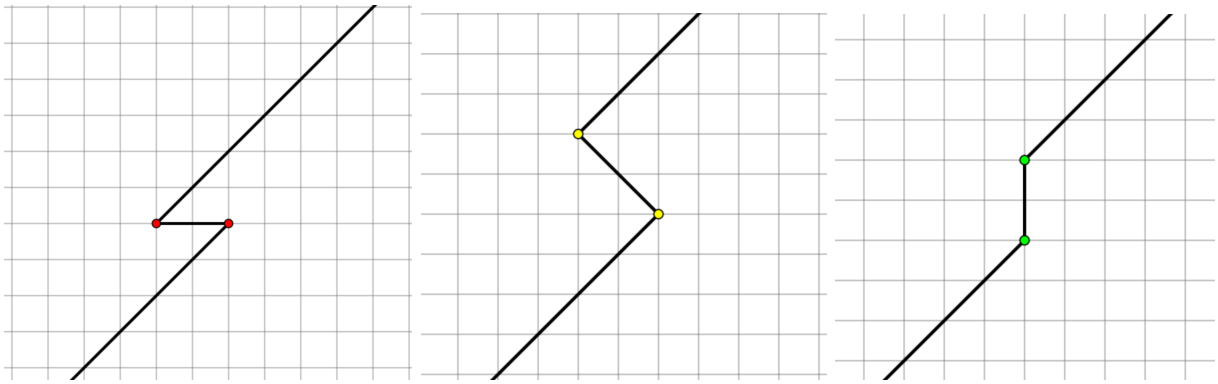
Author: Alexandru Gheorghies

Developer: Alexandru Gheorghies

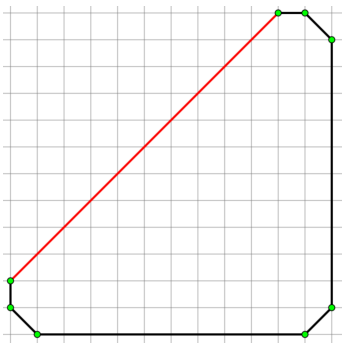
Solution

If a solution exists for (a, b, c), then a solution exists for $(a + 2 \cdot x, b + 2 \cdot y, c + 2 \cdot z)$, for all $x, y, z \in \mathbb{N}$.

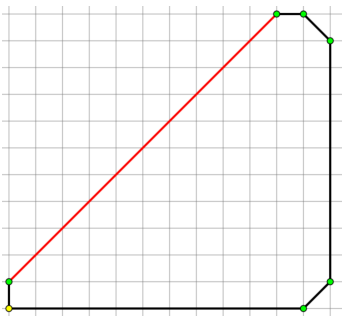
Suppose that we have constructed a polygon for (a, b, c) such that at least one of its edges is parallel to the line $x = y$ and sufficiently large. We can "insert" two turns of the same type in that edge:



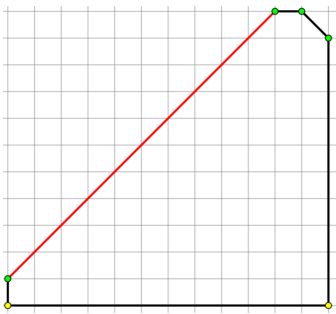
There are exactly 14 base cases, which are depicted below. The edge that will receive all of the extra turns is colored in red.



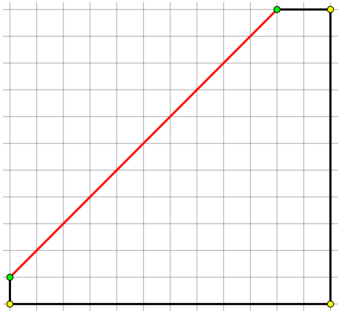
(0,0,8)



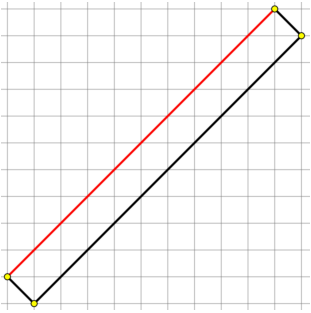
(0,1,6)



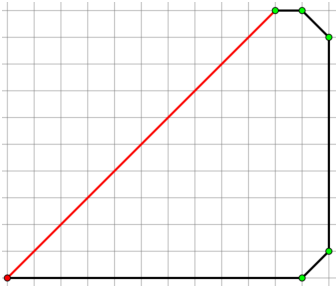
(0,2,4)



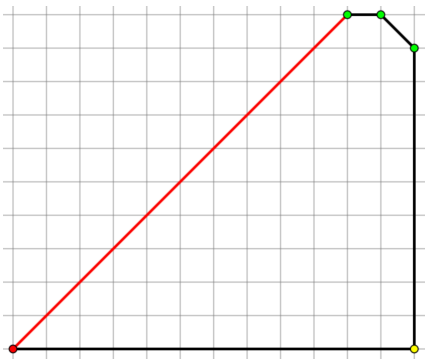
(0,3,2)



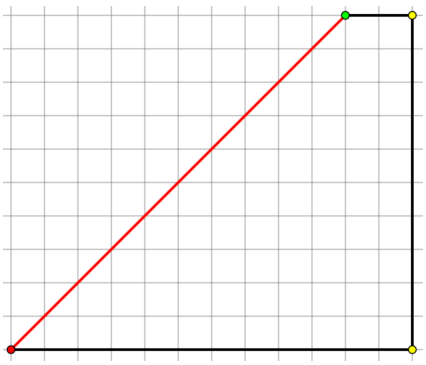
(0,4,0)



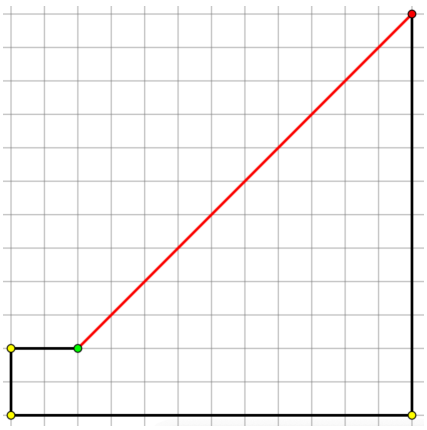
(1,0,5)



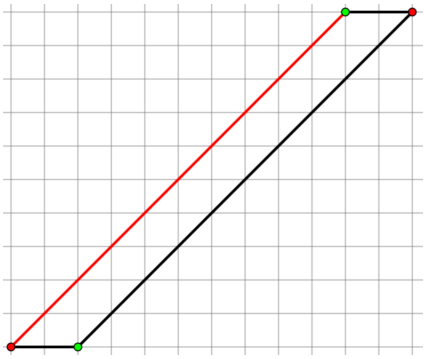
(1,1,3)



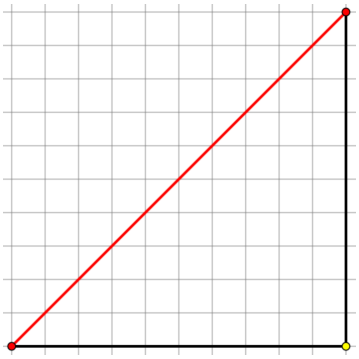
(1,2,1)



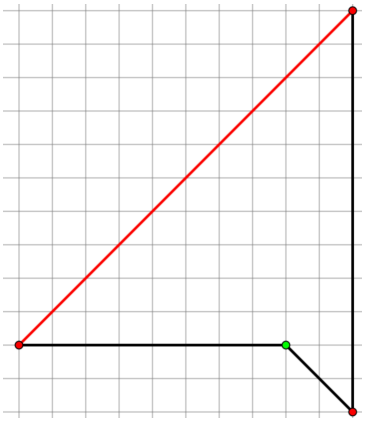
(1,3,1)



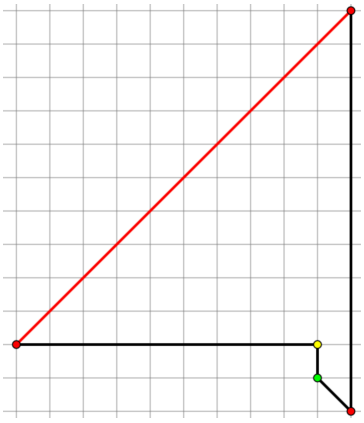
(2,0,2)



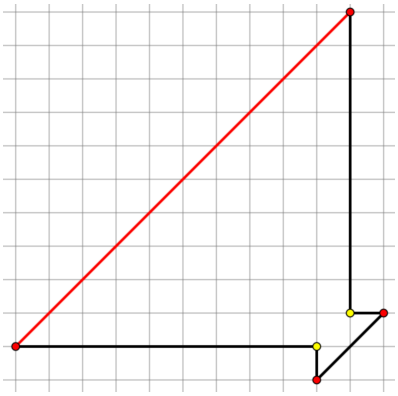
(2,1,0)



(3,0,1)



(3,1,1)



(4,2,0)

Time complexity: $O(a+b+c)$ per test case.

Meilì (meili)

Author: Luca-Stefan Camburu

Developer: Luca-Stefan Camburu

Solution

To approach this problem, we need to first understand $\nabla(N, M)$. We will try to find a formula for this quantity.

To study the parity of the *chic* subarrays, we could study the partial sums array of the initial array.

Let $V[i]$ be an array of N elements from $\{0, 1, \cdots, M\}$, indexed by 1. Let $S[i]$ be the array of the partial sums.

An subarray has an even sum if $S[j] - S[i-1]$ is even, where $1 \leq i \leq j \leq N$. More exactly,

$$S[i-1] \equiv S[j] \mod 2.$$

The number of subarrays with even sum is given by the number of elements in S that have the same parity.

We can now define

$$c_0 = \text{No. of even elements from } S;$$
$$c_1 = \text{No. of odd elements from } S.$$

Because the subarrays could start on the first position, we need $S[0], S[1], \cdots, S[N]$ to particularize all the subarrays.

Thus $c_0 + c_1 = N + 1$. Hence, the number of subarrays with even sum now becomes

$$L := \binom{c_0}{2} + \binom{c_1}{2},$$

and $L \equiv 0 \mod 2$.

Because we know $c_0 + c_1 = N + 1$, we could rewrite L as

$$\begin{aligned} L &= \frac{c_0(c_0-1)}{2} + \frac{(N+1-c_0)(N-c_0)}{2} \\ &= \frac{c_0^2 - c_0 + (N+1)N - (N+1)c_0 - c_0 N + c_0^2}{2} \\ &= \frac{2\,c_0^2 - c_0(1+N+1+N) + N(N+1)}{2} \\ &= \frac{2\,c_0^2 - c_0(2N+2) + N(N+1)}{2} \end{aligned}$$

$$\boxed{L = c_0^2 - c_0(N+1) + \frac{N(N+1)}{2}}.$$

From now on, L is only dependent on the parity of N, thus we can treat 4 cases, based on the remainder mod 4:

1. $N = 4k$, for some $k \in \mathbb{N}$. Then L becomes

$$\begin{aligned} L &= c_0^2 - c_0(4k+1) + \frac{4k\,(4k+1)}{2} \\ &= c_0^2 - c_0(4k+1) + 2k\,(4k+1) \\ &= c_0(c_0-1) - c_0\,4k + 2k\,(4k+1) \\ &= c_0(c_0-1) + 2\,(c_0\,2k + k\,(4k+1)) \end{aligned}$$

   Because $c_0(c_0-1)$ is always even, no matter what c_0, L is always even.

   So all arrays of $N = 4k$ elements from $\{0, 1, \ldots, M\}$ satisfy the condition.

2. $N = 4k+2$, for some $k \in \mathbb{N}$. Then L becomes

$$\begin{aligned} L &= c_0^2 - c_0(4k+3) + \frac{(4k+2)(4k+3)}{2} \\ &= c_0^2 - c_0(2\,(2k+1)+1) + (2k+1)(4k+3) \\ &= c_0^2 - 2\,c_0(2k+1) - c_0 + (2k+1)(4k+3) \\ &= c_0(c_0-1) - 2\,c_0(2k+1) + (2k+1)(4k+3) \end{aligned}$$

   We have that $c_0(c_0-1)$ is always even, as well as $2\,c_0(2k+1)$, but $(2k+1)(4k+3)$ is always odd, hence L is always odd.

   So all arrays of $N = 4k+2$ elements do not satisfy the condition.

3. $N = 4k + r$, for $k \in \mathbb{N}$ and $r \in \{1, 3\}$. Similarly:

$$\begin{aligned} L &= c_0^2 - c_0(4k+r+1) + \frac{(4k+r)(4k+r+1)}{2} \\ &= c_0^2 - 4k\,c_0 - c_0(r+1) + \left(2k + \frac{r+1}{2}\right)(4k+r). \end{aligned}$$

   We know that r is always odd, so $r+1$ is even, Thus, the parity of L is determined only by c_0^2 and $\left(2k + \frac{r+1}{2}\right)(4k+r)$. We write

$$L \equiv c_0^2 + \left(2k + \frac{r+1}{2}\right)(4k+r) \mod 2.$$

   If $r = 1$, then $L \equiv c_0^2 + (2k+1)(4k+1) \equiv 0$, thus c_0^2 is always odd, hence c_0 is odd.

   If $r = 3$, then $L \equiv c_0^2 + (2k+2)(4k+1) \equiv 0$, thus c_0^2 must be even, so c_0 is even.

To solve the last case, we can try to approach it using dynamic programming.

We thus define

$d[i][0/1][0/1]$ = Number of arrays of i elements that generate an even/odd number of even elements in S, and the parity of the last element from $S[i]$ is 0 or 1.

To ease the writing, we will call

$$\begin{cases} P = 1 + \lfloor \frac{M}{2} \rfloor & \text{(no. of even numbers} \leq M) \\ I = \lfloor \frac{M+1}{2} \rfloor & \text{(no. of odd numbers} \leq M) \end{cases}$$

The dynamic programming satisfies the following recurrences:

$$\begin{cases} d[i][\star][0] = d[i-1][\neg\star][0] \cdot P + d[i-1][\neg\star][1] \cdot I \\ d[i][\star][1] = d[i-1][\star][1] \cdot P + d[i-1][\star][0] \cdot I \end{cases}$$

By $\star$, we denote $\{0, 1\}$ and $\neg\star$ its "logical negation" (from odd to even, from even to odd).

Now, one can see that

$$\nabla(N, M) = \begin{cases} d[N][0][0] + d[N][0][1] & \text{if } r = 3 \\ d[N][1][0] + d[N][1][1] & \text{if } r = 1 \end{cases}$$

Computing the dynamic programming takes $O(N)$ time, thus one can naively compute $\nabla(N, i)$ for $i = 0, 1, \ldots, M$ and get an $O(NM)$ solution for the problem.

To further optimize this, we can explicitly solve for $\nabla(N, M)$ from the above recurrences.

For simplicity, we will denote only the parity states of the dynamic programming, based on the previous states. Concretely, this becomes

$$\begin{cases} (0,0) \leftarrow P \cdot (1,0) + I \cdot (1,1) \\ (1,0) \leftarrow P \cdot (0,0) + I \cdot (0,1) \\ (1,1) \leftarrow P \cdot (1,1) + I \cdot (1,0) \\ (0,1) \leftarrow P \cdot (0,1) + I \cdot (0,0) \end{cases}$$

For the answer, one only needs $(0,0) + (0,1)$ or $(1,0) + (1,1)$. Thus, we can sum and get

$$\begin{cases} (0,0) + (0,1) \leftarrow P((1,0) + (0,1)) + I((0,0) + (1,1)) \\ (1,0) + (1,1) \leftarrow P((0,0) + (1,1)) + I((0,1) + (1,0)) \end{cases}$$

We need another two sums, so we will have to also write them down. We get

$$\begin{aligned} (1,0) + (0,1) &\leftarrow P((0,0) + (0,1)) + I((0,1) + (0,0)) \\ &\leftarrow (P + I)((0,1) + (0,0)) \\ &\leftarrow (M + 1)((0,1) + (0,0)) \\ (0,0) + (1,1) &\leftarrow P((1,0) + (1,1)) + I((1,0) + (1,1)) \\ &\leftarrow (M + 1)((1,0) + (1,1)) \end{aligned}$$

This tells us that we are concretely going two steps back and we get to the same values.

Let $A_n = d[n][0][0] + d[n][0][1]$.

Let $B_n = d[n][1][0] + d[n][1][1]$.

We then have

$$\begin{cases} A_n = P(P+1)A_{n-2} + I(P+1)B_{n-2} = (M+1)(PA_{n-2} + IB_{n-2}) \\ B_n = P(P+1)B_{n-2} + I(P+1)A_{n-2} = (M+1)(PB_{n-2} + IA_{n-2}) \end{cases}$$

We will only work now with A_n and we will get $B_n = (M+1)^n - A_n$ (summing them up gives the number of all arrays).

Suppose now that M is odd. We get that $P = 1 + \frac{M-1}{2} = \frac{M+1}{2}$ and $I = \frac{M+1}{2} = P$.

One then gets that

$$A_n = (M+1)\frac{M+1}{2}(A_{n-2} + B_{n-2}) = \frac{(M+1)^2}{2}(A_{n-2} + B_{n-2}) = \frac{(M+1)^n}{2}$$

and $B_n = A_n$.

Suppose now that M is even. Then $P = 1 + \frac{M}{2}$ and $I = \frac{M}{2}$.

The recurrence becomes

$$\begin{aligned} A_n &= (M+1)\left(\frac{M}{2}(A_{n-2} + B_{n-2}) + A_{n-2}\right) \\ &= (M+1)\left(\frac{M}{2}(M+1)^{n-2} + A_{n-2}\right) \\ &= (M+1)^{n-1}\frac{M}{2} + (M+1)A_{n-2} \end{aligned}$$

From this form, one can write $n \to n-2 \to n-4 \to \ldots \to 1$ and plug it in the formula for n progressively. One gets that

$$A_n = \frac{M}{2}\sum_{i=1}^{\frac{n-1}{2}}(M+1)^{n-i} + (M+1)^{\frac{N-1}{2}}A_1 \qquad \left(A_1 = P = 1 + \frac{M}{2}\right)$$

$$A_n = \frac{M}{2}\frac{(M+1)^n - (M+1)^{\frac{n+1}{2}}}{M} + (M+1)^{\frac{n-1}{2}}\left(1 + \frac{M}{2}\right)$$

$$A_n = \frac{(M+1)^n - (M+1)^{\frac{n+1}{2}} + (M+1)^{\frac{n-1}{2}}(2+M)}{2}$$

$$A_n = \frac{(M+1)^n + (M+1)^{\frac{n-1}{2}}}{2}$$

Similarly,

$$B_n = \frac{(M+1)^n - (M+1)^{\frac{n-1}{2}}}{2}$$

We can now finally approach the initial problem. We summarize

$\nabla(N, M) =$

$$\begin{cases} (M+1)^N & \text{if } N \in 4\mathbb{N} \\ 0 & \text{if } N \in 4\mathbb{N}+2 \\ \frac{(M+1)^N}{2} & \text{if } M \text{ is odd and } N \text{ is odd} \\ \frac{(M+1)^N - (M+1)^{\frac{N-1}{2}}}{2} & \text{if } M \text{ is even and } N \in 4\mathbb{N}+1 \\ \frac{(M+1)^N + (M+1)^{\frac{N-1}{2}}}{2} & \text{if } M \text{ is even and } N \in 4\mathbb{N}+3 \end{cases}$$

To solve the sum (for N odd), one can split it in even-odd parts. Then it becomes

$$\sum_{i=0}^{M} \nabla(N,i) = \frac{1}{2}\left(\sum_{i \text{ odd}}^{M} (i+1)^N + \sum_{i \text{ even}}^{M} (i+1)^N \pm \sum_{i \text{ even}}^{M} (i+1)^{\frac{N-1}{2}}\right)$$

$$\sum_{i=0}^{M} \nabla(N,i) = \frac{1}{2}\left(\sum_{i=0}^{M} (i+1)^N \pm \sum_{i \text{ even}}^{M} (i+1)^{\frac{N-1}{2}}\right)$$

The $\pm$ symbol is based on the $4k+3$ or $4k+1$ cases.

We hence need sums of the form $2^n + 4^n + 6^n + \ldots$ and $1^n + 3^n + 5^n + \ldots$. One can solve this type of sum using Lagrange interpolating polynomial (for further reference, see this or this). One can first reduce the odd-base sum to the even-base sum using $(1^n + 2^n + \ldots) - (2^n + 4^n + \ldots)$. To solve the even-base sum, one notices

$$2^n + 4^n + \ldots + (2k)^n = 2^n(1^n + 2^n + \ldots + k^n)$$

So one can again reduce it to the simple $\sum_{i=1}^{m} i^n$, but with its range halved.

Time complexity: $O(N \log \text{MOD})$.