Problem Tutorial: "(-1,1)-Sumplete"

First, notice that for entries with the number 1, one can leave a 0 (cross it out) or an 1 (keep it). For entries with the number -1, one can leave a 0 (cross it out) or an -1 (keep it). By adding ones to the corresponding row/column sums for those -1 entries and flipping the result, the problem can be reduced to solving a Sumplete instance where all entries are filled with 1s.

We first safely assume that after the reduction, the required sum of each row and column is between 0 and n. Then, the following greedy algorithm works: Determine which entries are kept for each row in order. Suppose the current row has a required sum of s_i, then keep the entries at the columns with the s_i **largest** required sums. The original puzzle has no solution if, at any step, we are not able to find s_i columns with a positive required sum. It is not hard to prove the correctness of this algorithm. Note that this algorithm requires sorting the required sum of columns when processing the rows. Directly implementing this sorting takes $O(n^2 \log n)$, which is already enough to pass this problem. However, note that the required column sums are between 0 and n. So one can solve this problem in $O(n^2)$ by using, for example, counting sort.

Problem Tutorial: "Basic Equation Solving"

For each constraint in the form of $X < Y$ or $X > Y$, enumerate the highest different digit of X and Y. Then we get some constraints between 'letter and letter' or 'letter and number'. Use DSU to merge the letters with 'equal' constraints, then for the 'not equal' constraints such as $A < B$, add a direct edge from A to B. We need to count the number of ways to assign a number in $[0, 9]$ for each node without violating the constraints. Then we can independently calculate the number of solutions within each connected component.

Note that the number of 'not equal' constraints does not exceed 10, then each connected component has at most 11 vertices, let $dp(S, i)$ denote the number of ways that the vertices in S are assigned with number $0, 1, ..., i$. With some preprocessing, the dynamic programming can be computed in $O(10 \cdot 3^m)$ or $O(10m \cdot 2^m)$, where m is the number of vertices.

Under the restrictions given by the problem, the 'worst case' scenario is to give 10 constraints in the form of 'AB<CD'. In this case, the running time is $O(2^{10} \cdot 10 \cdot 3^{10})$, which is enough to pass the problem.

Problem Tutorial: "Bladestorm"

Let S denote the set of the health of all minions, and $dp(i)$ denote the minimum number of operations on the basis of already causing i damages to the entire field. The transfer of dynamic programming is $dp(i) = dp(\max(i + k, \min(j \mid j > i \wedge j \in S)) + 1$.

Consider the following graph construction: for each $0 \le i \le n$, if none of $i + 1, i + 2, ..., i + k$ is in set S, add an edge $i \to i + 1$ with weight 0, otherwise add an edge $i \to \min(n + 1, i + k)$ with weight 1. It can be shown that $dp(i)$ is equal to the distance from node i to node $n + 1$ in the graph constructed above.

In the process of adding $a_1, ..., a_n$ into set S sequentially, the graph maintains the tree structure, and there are n times of adding and deleting edges. We need to use data structures such as link-cut-tree or sqrt-decomposition to maintain the distance from node 0 to node $n+1$. The time complexity is $O(n \log n)$ or $O(n\sqrt{n})$ respectively.

Problem Tutorial: "Graph of Maximum Degree 3"

Note that the given graph has a maximum degree of at most 3. Therefore, for any induced subgraph with k vertices, there are at most $\frac{3k}{2}$ edges. Also, for a graph with k vertices to be connected by both red and blue edges, at least $2(k - 1)$ edges are needed. Since the inequality $\frac{3k}{2} \ge 2(k - 1)$ has no solution when $k > 4$, we can restrict ourselves to consider only the case for $k \le 4$.

Then, one can enumerate all connected induced subgraphs with only red edges with size ≤ 4 and check if they are also connected by blue edges. Again, due to the upper bound for the maximum degree, there are only $O(n)$ such induced subgraphs. Extra care is needed not to overcount the same subset of vertices more than once. The intended time complexity is $O(n)$ or $O(n \log n)$, depending on the implementation.

Problem Tutorial: "Inverse Topological Sort"

We first need to know what partial orders are required. For any $i < j$ such that $a_i > a_j$, we know that there must be a chain from vertex a_i to vertex a_j in G. The similar holds for any $i < j$ such that $b_i > b_j$. However, directly adding all these edges into the graph would lead to a quadratic amount of edges. To reduce the number of edges, we construct our graph G as follows:

- For any $1 \le j \le n$, add an edge (a_i, a_j) for the **largest** $1 \le i \le j$ such that $a_i > a_j$.
- For any $1 \le j \le n$, add an edge (b_i, b_j) for the **largest** $1 \le i \le j$ such that $b_i < b_j$.

Clearly, this construction of G satisfies all the partial orders with at most $2n-2$ edges without introducing any additional partial orders. Hence, this construction is valid as long as at least one solution exists.

To check if this construction of G is actually valid, we still need to compute the lexicographically smallest/largest topological ordering of G and compare them with the two arrays. A way of doing so is to modify Kahn's algorithm by maintaining all valid vertices in a priority queue, each time choosing the next vertex greedily. The overall complexity would be $O(n \log n)$.

Problem Tutorial: "Land Trade"

Each atomic formula corresponds to a straight line on the plane, the n lines divide the rectrangle area into at most $O(n^2)$ polygons, where the total number of edges of all the polygons is $O(n^2)$. In order to find these polygons, we can add the straight lines one by one and maintains all the current polygons. The time complexity is $O(n^3)$.

Then for each polygon, we arbitrarily select a point inside the polygon (such as the centroid) and check whether it satisfies the clause, if so, we add the area of the polygon to the answer. It takes $O(n)$ times for each polygon, then the total complexity is $O(n^3)$.

Problem Tutorial: "Parity Game"

First, we need to determine who will win the game. We call the player who will make the last move *the Last Player*, or simply, LP, and the other player *the Second Player*, or simply, SP. We consider two cases. The first case is when LP needs the final number to become 0 to win, which we call the "zero case', and we call the other case the "one case".

Lemma G.1 *If the game is under the zero case, then LP always wins.*

The proof to Lemma G.1 is direct since no matter what two numbers are left, LP can always make them become 0 in the last move. The slightly harder case is the one case.

Lemma G.2 *If the game is under the one case, then LP wins if and only if:*

- After shrinking all consecutive occurrences of 1 to the 0/1 number matching its parity (i.e., shrinking an odd number of consecutive 1s into one 1 and even ones into nothing), the number of 1s is at least the number of 0s.

Lemma G.2 can be easily verified using induction. We briefly discuss how to come up with it besides brute-forcing the small cases and observing the pattern. Note that LP loses only when the last two numbers are both 0 in the last move. So LP will try to maximize the difference between 1s and 0s, and SP will try to minimize it. Normally, one step will change the difference by at most 1 except for the case when merging two 1s into a 0 using addition. Considering the effect of this case gives the lemma above.

To actually play the game interactively, since the constraint is low, one is allowed to enumerate all possible choices and compute the winner of the game. This takes $O(n^3)$ time.

Problem Tutorial: "Random Tree Parking"

Given a rooted tree T and a vertex $v \in V(T)$, we let $\mathrm{Sub}_T(v) \subseteq V(T)$ denote the subset of vertices in the subtree rooted at v. We first need to make the following observation.

> **Lemma H.1** *Given any tree T with n vertices and a sequence $\mathbf{s} \in [n]^n$, $\mathbf{s}$ is a parking function of T if and only if the following holds:*
>
> - For all $v \in V(T)$, $|\{1 \le i \le n \mid \mathbf{s}(i) \in \mathrm{Sub}_T(v)\}| \ge |\mathrm{Sub}_T(v)|$

We first prove Lemma H.1. The only-if direction is simple since an insufficient amount of starting cars in a subtree will obviously leave an empty space in that subtree. To show the if direction, note that a sufficient amount of starting cars in some subtree means that the root of this subtree will eventually be occupied. Then, the condition in Lemma H.1 implies that all vertices will be occupied, meaning $\mathbf{s}$ is indeed a parking function.

We remark that Lemma H.1. also reveals that whether some $\mathbf{s} \in [n]^n$ is a parking function or not does not depend on the order of the elements.

Then, one only needs to consider for each vertex how many times it is present in the function. we can try to solve the problem using the following dynamic programming: Let $dp_{v,i}$ denote the number of ways to assign this "occurrence time" and arrange them for each vertex in the subtree of v so they are valid and sum to i.

Calculating this dynamic programming would naturally take $O(n^3)$ time. However, note that in this dynamic programming, for each vertex v, the valid choices of second dimension i can only range from $|\mathrm{Sub}_T(v)|$ to $|\mathrm{Sub}_T(v)| + \mathrm{dep}_T(v)$, where $\mathrm{dep}_T(v)$ means the depth of v in the tree T. (The depth of the root is 0). The lower bound is by Lemma H.1, and the upper bound is also easy to observe. (If the amount of starting cars in the subtree of v exceeds $|\mathrm{Sub}_T(v)| + \mathrm{dep}_T(v)$, surely some car has no place to park since they can only park in the subtree of v and in the path between the root and v.) Then, this dynamic programming can be calculated in $O(nh^2)$ time, where h is the height of the tree.

This is where the condition that the tree is generated randomly comes into play, that the generated tree with high probability has height $O(\log n)$. One can calculate that with probability no more than 10^{-9} (which we can safely assume will not happen), this random process will generate a tree with a height of more than 60. Therefore, this problem is solved in $O(n \log^2 n)$ time.

Problem Tutorial: "Refresher into Midas"

It's easy to see that one should always use the Hand of Midas as long as it's ready. Also, one should always use the Refresher Orb when it's ready, and you have just used the Hand of Midas. So initially, you should Midas, Refresh, and then Midas, leaving both in cooldown. Let f_t denote the maximum number of gold you can gain when both the Hand of Midas and the Refresher Orb are in cooldown just now. There are generally two possibly optimal strategies:

1. Use the Refresher Orb as long as it's ready, i.e., after b seconds, gain $160 \times \left(\lfloor \frac{b}{a} \rfloor + 1\right)$ gold.
2. Use the Refresher Orb after use of the Hand of Midas, i.e., after $a \times \lceil \frac{b}{a} \rceil$ seconds, gain $160 \times \left(\lceil \frac{b}{a} \rceil + 1\right)$ gold.

Then, the overall time complexity is $O(m)$.

Problem Tutorial: "Teleportation"

This is a shortest path problem with unit edge weights. A direct approach to this problem is to use a breadth-first search. However, this would include $O(n^2)$ edges. To reduce the complexity, first observe that in an optimal solution, one never passes the same room more than once. Then, one can consider the

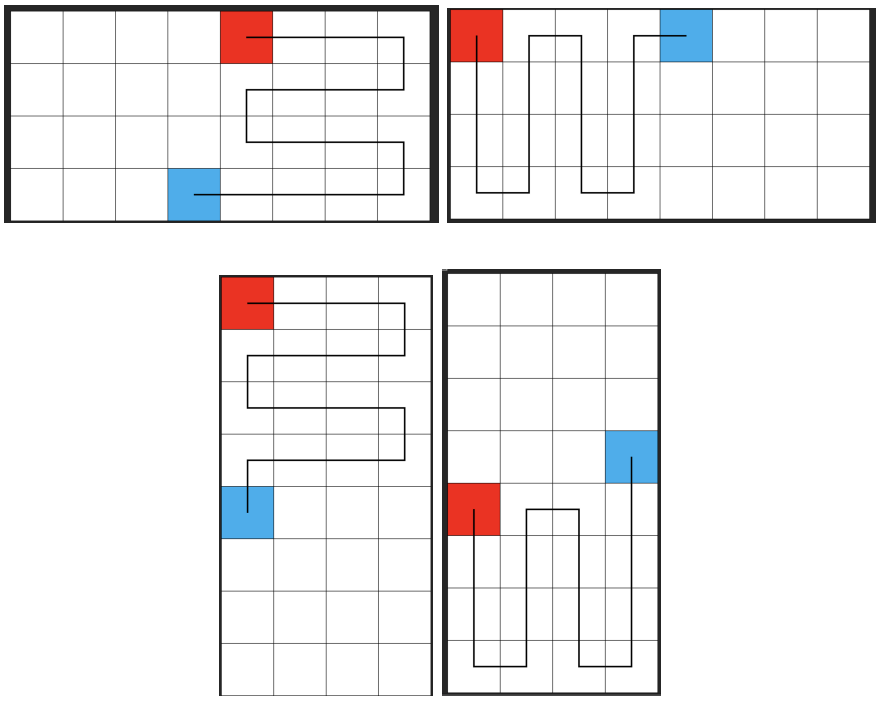
second operation (move the hand clockwise) as "deferred"; that is, after performing the first operation at least once, one has the choice of "go from room i to $i+1 \bmod n$". Then, the number of vertices in this new graph becomes $n+1$, and the total number of edges becomes $2n-1$ at most.

Problem Tutorial: "Understand"

First, we find that as long as we construct 16 paths so that for each cell, the subset of the 16 lines that pass it is different, then the problem is solved.

We call the division of the entire table into $2^k \times 2^k$ squares with a side length of 2^{8-k} as the k-level division ($0 \le k \le 8$). We then try to construct a group for each $1 \le i \le 8$, each containing two paths (which we call answer paths), so that in each square of the i-level division, all cells are passed by the same set of paths, and in each square of the $(i-1)$-level division, the four squares of the i-level division have different passing situations.

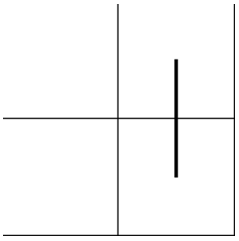
Note that starting from the upper left corner or lower right corner of a square with an even side length, it is always possible to traverse all the cells and reach one of the upper left corner/right corner cells of the same-sized square in any of the four directions (as shown in the figures below).



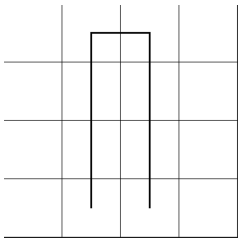
We just need to construct a set of paths (which we call generating paths) that satisfy the condition "in the scenario of a square with a side length of 2^i, all cells in each square of the i-level division are passed by the same set of paths, and the four squares of the i-level division in each square of the $(i-1)$-level division have different passing situations" to obtain the i-th group of answer paths that satisfy the original requirement.

Because the requirements for each generated path group are locally consistent, we can construct them recursively.

In our construction, each group's generated paths are symmetric with respect to the diagonal from the top left to the bottom right. We will only show the first path in each group. The first path of the first group is shown in the figure below:

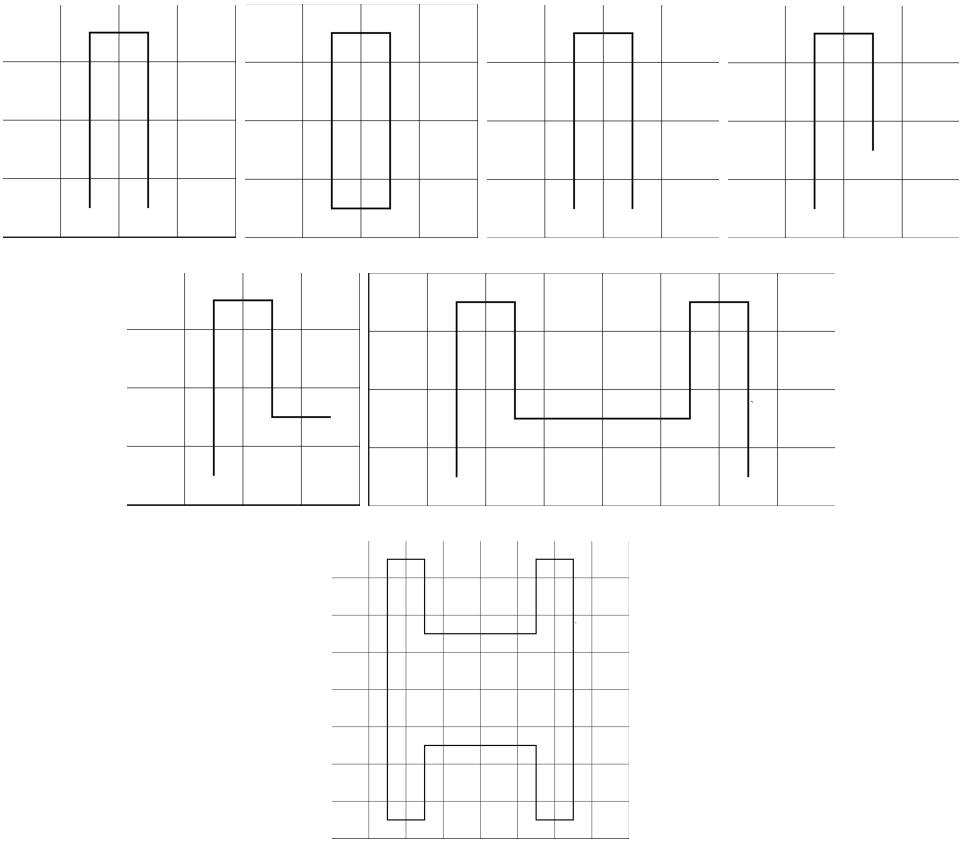


The first path of the second group is shown in the figure below:

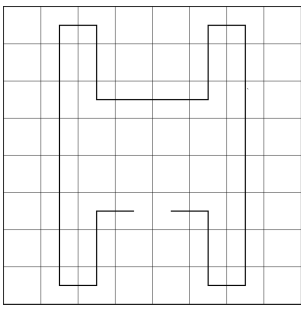


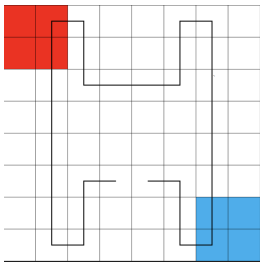
It can be noticed that this path not only meets the requirements for generating paths but also can be connected at both endpoints to form a loop, with the left half occupying the bottom right corner of the grid. For the first line of the generated path, we generate it based on the first line of the $(i-1)$-th generated path. The resulting path still meets the requirements for generating polylines and connects at both ends to form a loop, with the left half occupying the bottom right corner of the grid. Our generation method is as follows:

First, connect the paths into a loop, then delete the second-to-last cell in the last row. Now, the loop is broken into a path, with the two endpoints in the third-to-last cell of the last row and the second-to-last cell of the second-to-last row. We extend the second-to-last cell of the second-to-last row to the right by one cell. Now, the two endpoints of this path are in the third-to-last cell of the last row and the last cell of the second-to-last row. We then perform a reflection with the rightmost side as the axis of symmetry, followed by a reflection with the bottom side as the axis of symmetry.

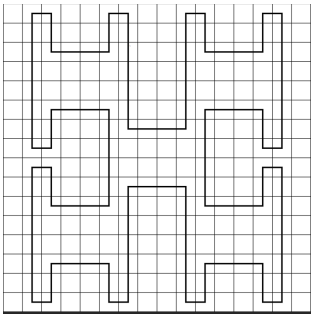


Now, the generated figure is a loop. If it is disconnected from any point, a path that not only meets the requirements of generating paths but also forms a loop when the two endpoints are connected is obtained. This is because the square grid at the bottom right is obtained by reflecting the square grid at the top left twice along the axis, and the area where the path passes through in the top left square grid remains unchanged. Therefore, the path occupies the left half of the square grid at the bottom right.

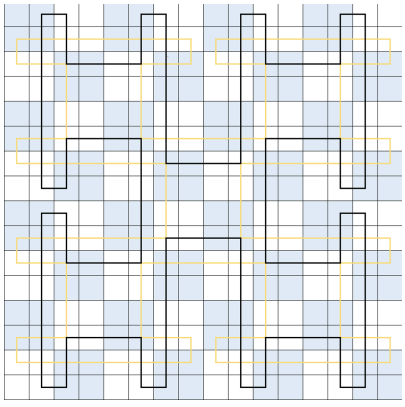




The following image shows the first line of the fourth group generated from the first path of the third group using the same method (connecting the endpoints of the path to form a loop).



The following image shows the full fourth group (connecting the endpoints of the path to form a loop).



We can use these generated paths to generate the answer path.

We ask about these 16 paths and record the situation of each cell being crossed by the 16 answer paths. This will give us the answer to the original problem.