Problem A. Another GCD Problem

- Considering the direct computation of $f(k)$, we can find that $f(k) = \gcd(ka_1, a_2 - a_1, \ldots, a_n - a_1)$.
- Therefore, if all elements in a are equal, the answer is infinite. Otherwise, the answer is $ans = \gcd(a_2 - a_1, \ldots, a_n - a_1)$, and the smallest k is $\mathrm{lcm}(a_1, ans)/a_1$.
- Derivation of the value of $f(k)$: Let $t = f(k)$. Clearly $t \mid \gcd(a_2 - a_1, a_3 - a_1, \ldots, a_n - a_1)$. Also, clearly $t \mid ka_1$, thus proven.
- Sufficiency follows from $\gcd(a, b) = \gcd(a, a + b)$.
- Time complexity is $O(n \log n)$.

Problem B. Ball And Movable Walls

A particle is inside a two-dimensional rectangle, given its initial velocity. The sides perpendicular to the x-axis expand outward at a certain speed, while the sides perpendicular to the y-axis compress inward at a certain speed. Upon colliding with any side, the component of the particle's velocity perpendicular to that side is completely reversed. Determine the position of the particle after the compression ends.

- Consider v_x and v_y separately. The change in the y-coordinate is straightforward, and the end time T_{end} can be calculated.
- If the speeds of the left and right plates are both 0, the answer can be calculated directly.
- Otherwise, the particle will only collide in the left-right direction $O(\log(T_{\text{end}}))$ times. This is because the distance traveled by the particle from one collision to the next increases by at least 1/20 each time.
- Therefore, collisions in the left-right direction can be simulated by brute force.

Time complexity per test case: $O(\log(T_{\text{end}}))$.

Problem C. Covering Problems

In the most basic version, it can be assumed that the operations have no intersecting relationships other than containment (except for endpoint intersections).

Conclusion: The optimal cost is $\sum_{i=1}^{n} \max(a_i - a_{i-1}, 0) b_i + \max(a_i - a_{i+1}, 0) c_i$.

- This is actually very easy to understand: the number of intervals with i as the left endpoint must be at least $\max(a_i - a_{i-1}, 0)$, and similarly for the right endpoint. Since a scheme achieving exactly this can be constructed, this is indeed the optimal solution.
- The answer to the second problem is essentially the same as that of the first problem. This is because if $\max(a_i - a_{i-1}, 0) > d_i$ or $\max(a_i - a_{i+1}, 0) > e_i$, then continuously adding this point would make the answer infinite. Otherwise, it certainly does not need to be added.
- Thus, the counting becomes very straightforward. Let $f_{i,j}$ denote the number of valid schemes where $a[i, n], b[i + 1, n], c[i, n], d[i + 1, n], e[i, n]$ are determined and $a_i = j$. Let $g_{i,j}$ denote the sum of the corresponding weights for all valid schemes. The transitions can be optimized using prefix sums to achieve $O(nV)$ complexity.

Problem D. Director and Excitement

Brief Description

Given two increasing arrays l and r of length n, and a fixed constant k.

There are q queries. Each query provides an interval $[L, R]$ and a constant D, asking whether it is possible to divide these people into k groups such that within each group, there exists a way to assign $x_i \in [l_i, r_i]$ so that the minimum value of $|x_i - x_j| (i \neq j)$ within the group is no less than D.

Assume $l_i = r_i$. It can be found that the grouping must be: $[L, L+k, L+2k, \ldots], [L+1, L+1+k, L+1+2k, \ldots], \ldots$. That is, positions with the same modulo k are assigned to the same group.

Assume $k = 1$. The problem is equivalent to checking whether $\forall i < j, (r_j - l_i)/(j - i) \geq D$.

In summary, each query essentially asks whether for all $i < j$ within the interval that have the same modulo k, the condition $r_j - jD \geq l_i - iD$ holds. This form can be maintained by various data structures. The standard implementation is described below.

Consider a static divide-and-conquer structure similar to a segment tree. For a node x representing the interval $[l, r]$, precompute val_x which denotes the maximum feasible D within $[l, r]$.

- val_x can be obtained by first taking the minimum of val_{ls_x} and val_{rs_x}, then constructing convex hulls for $[l, mid]$ and $[mid + 1, r]$ respectively and performing queries.
- If binary search is used for tangent line queries on the convex hull, the time complexity for this part is $O(n \log n \log D)$. If two-pointers are used, it becomes $O(n \log n)$.
- For a query $[L, R]$, similar to segment tree queries, we can insert all queries into nodes where the interval straddles the midpoint (i.e., $l \leq L \leq mid < R \leq r$). When recursively reaching a whole interval, directly return val_x.
- For queries inserted at a particular node, since D is fixed, we can separately compute the convex hulls for $[l, mid]$ and $[mid + 1, r]$, query for the extreme values at slope $k = D$, and then compare these extreme values.
- Each query is inserted into $O(\log n)$ nodes, and each node requires a binary search for the extreme value. Therefore, the time complexity for this part is $O(qk \log^2 n)$.

The total time complexity is $O(n \log n + qk \log^2 n)$ or $O(n \log n \log W + qk \log^2 n)$. In practice, the time limit is relatively lenient, and any solution with logarithmic complexity should be acceptable.

Problem E. Fun With Bitwise OR

Given an array containing only 1 and 2, find the number of essentially distinct sequences under adjacent bitwise OR operations. It is guaranteed that the lengths of consecutive identical digit blocks are non-decreasing.

- Consider a subsequence automaton. That is, for a sequence, we want to find the shortest prefix such that it can be transformed into that sequence via operations. Then consider the new shortest prefix after appending a number to this sequence.
- If the current sequence contains no 3, the transitions are fixed. If currently at the end of a 1 (or 2) block, you cannot append 1 (or 2). Otherwise, appending 1 moves to the next position in the same block, and appending 2 or 3 moves to the start of the next block.
- If the current sequence contains a 3, since a 3 can delete any subsequence after it, even if currently at the end of a 1 (or 2) block, you can append 1. This transitions to the same position or the next position in the next block of 1s.
- Since the block lengths are non-decreasing, this transition is valid.

- Based on this idea, write a DP. There are quite a few details.

Time complexity: $O(n)$.

Problem F. Great Snowfall

Given an $n \times m$ snow pile, snow can be shoveled to the right or downward at no cost. Alternatively, snow can be shoveled or added unconditionally at a cost of 1. Find the minimum cost to flatten all positions.

- Use a greedy approach from the top-left to the bottom-right. For a pile of snow, shovel it downward as much as possible under the premise that its right side can be flattened.
- A necessary and sufficient condition for a row of snow to be flattenable is that the sum of their heights is 0 and every prefix sum is non-negative.
- Therefore, maintain the suffix minimum of the prefix sums for each row. If this value is less than 0, additional snow must be added. Otherwise, the excess (positive part) can be transferred downward.

Time complexity is $O(nm)$ or $O(nm \log(nm))$, depending on the implementation. Both are expected to pass.

Problem G. Efficient Summation

The first approach was written by dls during problem testing. It is similar to the traditional min25 sieve, where the last prime number is handled separately during the search. The complexity is completely incalculable, but it passes. However, this approach is extremely difficult to write.

Let's provide a more reasonable approach: Consider storing all states during the min25 $n^{1-\epsilon}$ search and performing a high-dimensional suffix sum on them. However, you will find that this undercounts when the maximum prime factor exponent of the query x is 1. Thus, we can consider a sqrt-decomposition on the maximum prime factor of x. Let the threshold be B. For maximum prime factors $p \le B$, recompute the high-dimensional suffix sum each time. Otherwise, the problem can be transformed into a two-dimensional point query problem where one dimension is within n/B. By offline processing the larger dimension and maintaining it with a block data structure that supports $O(1)$ addition and $O(\sqrt{n})$ query, and since the other dimension has size $\sqrt{n}$, the query complexity per instance is $O(n^{1/4})$.

Choosing an appropriate B allows passing this problem. The time limit is set quite generously and does not emphasize constant optimization.

Problem H. How Many Rounds Are Expected?

Use the potential method to analyze. Each set only depends on its own size and is independent of the states of other sets. Thus, define $f(a)$ as the potential function for a set containing a elements.

- Let the current state have sets with element counts $b_1, b_2, \ldots, b_{n_0}$. All possible pairing schemes form a set S. After applying the i-th pairing scheme, the element counts of the sets become $b_1^i, b_2^i, \ldots, b_{n_i}^i$.
- Then we have $1 + \sum_{i=1}^{n_0} f(b_i) = \frac{1}{|S|} \sum_{i=1}^{|S|} \sum_{j=1}^{n_i} f(b_j^i)$. The answer to the original problem is $f(m) - \sum_{i=1}^{n} f(a_i)$.
- Solve for $f(a)$ using Gaussian elimination. When constructing equations for $f(a)$, it suffices to consider the change in the element count of a set containing a elements after one operation.

Problem I. Infinite Hexagonal Grids

Given a black-and-white hexagonal grid graph. In each operation, you can choose a hexagon and flip the colors of its adjacent hexagons (without flipping itself). Determine whether one grid can be transformed into another through a sequence of such operations.

- Conclusion: The answer is YES if and only if for each x coordinate (and similarly for each y coordinate and each z coordinate), the parity of the number of black cells is the same in both grids.
- Necessity: Obviously, a single operation does not change the parity of the number of black cells with the same x coordinate (or y or z coordinate). (From the diagram, each coordinate involves exactly two cells being flipped.)
- Sufficiency: The problem is essentially equivalent to transforming a given grid into an all-white grid (which can be achieved via an operation similar to XOR).
- Consider constructing the solution from the outermost layer inward, layer by layer. Each time, the outermost layer (where x, y, or z attains its maximum) also forms a hexagon.
- A straightforward construction method is: first flip the black cells at the six corners, then independently handle the black cells on the six edges. From the condition above, the number of black cells on each edge is even, so they can all be flipped to white.

Time complexity: $O(n \log n)$.

Problem J. Judge of Gensokyo

Given a string s consisting only of the characters '`o`', '`v`', and '`w`', where '`w`' is treated as '`vv`'. Find the longest palindromic substring of this string.

A natural idea is to first replace every '`w`' with two '`v`' (let the resulting string be s'), then run Manacher's algorithm.

This approach has two issues:

1. The longest valid substring in s' may not correspond to the longest valid substring in s.
2. A substring in s' might not correspond to any valid substring in s. For example, the substring "`wov`" in s becomes "`vvov`" in s'. Its longest palindromic substring might be "`vov`", but this does not represent any substring in s.

The first issue can be resolved by enumerating each palindrome center and finding the longest possible radius.

For the second issue, we can consider repeatedly removing '`w`' from both ends until the first non-`w` character is encountered at each end. This yields the longest palindromic substring in s that corresponds to the current palindrome center.

Thus, we enumerate each palindrome center and its maximum palindrome radius in s', and then repeatedly shrink the w's to find the longest palindromic substring that actually exists in s. Time complexity: $O(|s|)$.

Problem K. Knapsack And Greedy Algorithm

Given a W_{lim}, construct a 0/1 knapsack instance with as few items as possible, and with item volumes and values as small as possible, such that for every knapsack capacity W from 2 to W_{lim}, the greedy algorithm fails to produce the correct solution.

The unique construction method (assuming $n = W_{lim}$):

- Item volumes: $1, n, n-1, \ldots, 2$
- Item values: $n, n^2-1, n^2-n-2, \ldots, n+1$

First, prove that after sorting, the item volumes must be $1, n, n-1, \ldots, 2$.

For $n = 2$, the only feasible scheme is volumes $1, 2$ (the additional 1 after 1,2 is useless).

- Next, consider induction. Assume that for $n-1$, the unique scheme is volumes $1, n-1, \ldots, 2$. Then for capacity n, the greedy algorithm clearly obtains the optimal solution.
- Due to uniqueness, we cannot insert an item with volume $\leq n-1$. Hence, we can only insert n after 1, resulting in volumes $1, n, n-1, \ldots, 2$.

Next, consider the item values. The values must satisfy:

$$\begin{cases} \bullet\ \nu_1 > \frac{\nu_n}{n} > \frac{\nu_{n-1}}{n-1} > \ldots > \frac{\nu_2}{2}. \\ \bullet\ \forall 1 < i < n,\ \nu_i + \nu_1 < \nu_{i+1},\ \nu_1 < \nu_2. \end{cases}$$

It can be observed that, given the second condition and that item 1 is first, the first condition naturally holds. Therefore, we need to find the smallest ν_1 such that:

$$\nu_n = \nu_1 + 1 + (n-2)(\nu_1 + 1) < n \cdot \nu_1$$

which simplifies to $\nu_1 > n - 1$. Hence, $\nu_1 = n$. Then the values $\nu_2, \nu_3, \ldots, \nu_n$ can be derived accordingly.

Problem L. Let's Avoid Obstacles

Given an $n \times m$ grid with k obstacles. Consider a path that starts from some row in the first column and ends at some row in the m-th column, without passing through any obstacles. For each obstacle, there are two ways to pass it: from above or from below.

For all 2^k possible choices (one for each obstacle), compute the minimum path length.

- Simulate according to the description. Perform BFS while recording the passing state for each obstacle.
- State compression can be used to record the passing states of obstacles. Time complexity: $O(nm2^k)$.

Problem M. Multiple Triangular Intersections

Given a regular triangular grid of size n. All paths in each direction are identical. Count the number of connected equilateral triangles in the grid.

Assume an equilateral triangle is formed by three intersecting paths: one of length i in the right-diagonal direction, one of length j in the left-diagonal direction, and one of length k in the horizontal direction.

A necessary and sufficient condition for these three paths to determine an equilateral triangle is:

$$i + j \geq n, \quad j + k \geq n, \quad i + k \geq n.$$

(Otherwise, two paths do not intersect.) Additionally,

$$i + j + k \neq 2n.$$

(If $i + j + k < 2n$, it is an upright triangle; if $i + j + k > 2n$, it is an inverted equilateral triangle; if $i + j + k = 2n$, it degenerates to a point.)

First, ignore the second condition and count the number of triples satisfying the first condition. We can fix $i = \min(i, j, k)$ and enumerate i. Then we know the lower bounds for j and k, and the number of valid (j, k) can be computed directly using suffix sums.

Then, subtract the number of triples satisfying the second condition. Note that the second condition implies the first condition. Thus, FFT can be used to compute this count.

Time complexity: $O(n \log n)$.

Bonus: How $O(n)$? (Hint: The current approach cannot avoid the FFT step for the second condition, so one should try to count the invalid cases directly.)