Problem A. Atcoder Problem

It's a harder version of this Atcoder problem. You may read the editorial of the original problem first to get a brief idea of this problem. But to be honest, the solution in Atcoder's editorial is far from solving this problem.

The first solution

There are mainly three parts to solving this problem:

1. Solve the problem without the non-decreasing constraint.
2. Reduce the problem to the unordered version.
3. Use generating function tricks to speed up the second step.

The first part

In the first part, we try to solve the problem without the non-decreasing constraint.

You need to compute the number of sequences $A_i(0 \le A_i \le M)$, and $\bigoplus_{i=1}^{n} A_i = X$ for $n = 0, 1, 2, \dots, NMAX$.

Let $W = \lceil \log M \rceil$. Let S_i be the multiset that the xor of i numbers in $[0, M]$. We can see $S_{i+1} = \{x \oplus y | x \in S_i, 0 \le y \le M\}$. The answer is the number of occurrences of X in the set $S_1, S_2, \dots, S_{NMAX}$. So we try to maintain S_i directly in some way, and we can compute a convolution-like thing for S_i and the interval $[0, M]$ to get S_{i+1}.

We can partition S_i into several subsets. The numbers in each subset are in $[x \cdot 2^k, x \cdot 2^k + (2^k - 1)]$, and each number occurs c times. We can denote this subset by (x, k, c). A more intuitive way to understand this definition is that for each subset, the higher bits of these numbers are fixed, and the lower bits of these numbers are evenly distributed. And we call k as the length of this subset.

It's not hard to see we can partition the interval $[0, M]$ into $O(W)$ sets. We can show each S_i can be represented by $O(W)$ sets inductively.

When we want to compute the convolution of S_i and $[0, M]$, we can compute the convolution of each pair of subsets and add them up. For a pair of subsets (x_1, k_1, c_1) and (x_2, k_2, c_2)(WLOG, we assume $k_1 \ge k_2$ here), after the convolution, the lower k_1 bits will be evenly distributed, and the occurrence will become $c_1 \cdot c_2 \cdot 2^{k_2}$. The higher bit will be the higher bits of $(x_1 \cdot 2^{k_1}) \oplus (x_2 \cdot 2^{k_2})$. And we can also prove inductively, that for a fixed k, $x \in \{0, 1, \lfloor M/2^k \rfloor, \lfloor M/2^k \rfloor \oplus 1\}$. Then S_i can be represented by at most $4W$ subsets.

We get an $O(W^2)$ algorithm to compute the convolution of two multisets, which may not be fast enough. We can notice when we compute the convolution of a pair of subsets with length $k_1, k_2 (k_1 \ge k_2)$, only $M - k_1$ higher bits matter for the subset with length k_2. We can do the convolution in the increasing order of k, and all subsets with lengths smaller than k can be merged by the same $M - k$ higher bits, there will be $O(1)$ subsets after the merge. So we can get an $O(W)$ algorithm to compute the convolution. And we can solve this part in $O(NW)$.

We denoted the answer of i numbers by f_i.

The second part

In the second part, we try to reduce the original problem to the unordered version.

You can consider the problem in another way. You are still solving the problem without the non-decreasing constraint, but two solutions $A_1, A_2, \dots, A_n$ and $B_1, B_2, \dots, B_n$ are equivalent iff we can rearrange the order of A to get B, and find the number of the equivalence classes. It's not hard to see, the number of equivalence classes is equal to the number of solutions with the non-decreasing constraint.

Then we can use Burnside's lemma to count the number of equivalence classes. The answer equals to

$$\frac{1}{n!} \sum_{g \in S_n} (\#\text{number of solutions fixed by } g)$$

For a permutation $g \in S_n$, we can decomposite it to several cycles. If the number of solutions is fixed by g, A_i in one cycle should be the same. So for a cycle with an even length, all A_i will be canceled out, so we have $(m+1)$ choices for it. For a cycle with an odd length, there will be one A_i remained. Thus, for a permutation with p even cycles, and q odd cycles, the number of fixed solutions is $f_q \times (m+1)^p$.

We can use dynamic programming or other combinatorial approaches to compute how many permutations with p even cycles and q odd cycles, and add them up to get the answer. But the time complexity can be very high. However, we can compute the odd and even cases independently to get a better solution.

For even cycles, let e_n be the sum of the weight of permutations on n elements that only consist of even cycles, and the weight of the permutation is $(m+1)^{\#\text{cycles}}$. For odd cycles, let o_n be the sum of the weight of permutations on n elements that only consist of odd cycles, and the weight of the permutation is $f_{\#\text{cycles}}$.

If we get e_n and o_n, the answer for n is just $\frac{1}{n!}\sum_{i+j=n} e_i \times o_j \times \binom{n}{i}$, which is a simple convolution.

The third part

In the third part, we will use some generating function tricks to speed up the computation of e and o. Since the even case and the odd case are separated, you may get a better solution with dynamic programming, but it's still impossible to solve the case when $n = 10^5$ without some advanced techniques.

For the even case, it's much easier. It's not hard to see the EGF of an even cycle is $-\frac{1}{2}(\ln(1-x)+\ln(1+x))$. So the EGF of e is $\exp(-\frac{m+1}{2}(\ln(1+x)+\ln(1-x)))$, which can be solved easily in $O(n \log n)$.

For the odd case, it's more complicated. Let $ODD(x) = \frac{1}{2}(\ln(1+x) - \ln(1-x))$, which is the EGF of an odd cycle. Let $F(x)) = \sum_{i=0}^{n} f_i \frac{x^i}{i!}$, which is the EGF of the sequence f.

So $o_n = [x^n]\left(\sum_{i=0}^{n} f_i \frac{ODD^i}{i!}\right) = [x^n]F(ODD(x))$, which means you need to compute the composition of polynomial F and ODD. It's hard to compute the composition of two arbitrary polynomials. But ODD is very special here, so it's possible to compute it in $O(n \log^2 n)$.

There is a similar problem and detailed tutorial here. In that problem, you are asked to compute the composition of an arbitrary polynomial and the EGF of cycles with a length greater than 1. However, the solutions are almost the same. I will introduce the brief idea here. If you want to have a better understanding, it's recommended to start by learning how the transposition principle (or Tellegen's Principle) works in generating function manipulation. Here are some materials: 1 (in Chinese), 2 (in Chinses), 3 (in English).

We can define a $n \times n$ matrix $G_{n,k}$ be the number of permutations with n elements consisting of k odd cycles, and a $n \times 1$ matrix $v_{k,1} = f_k$. In this problem, we want to compute $G \cdot v$. However, $G \cdot v$ may be hard to compute directly, we can try to find how to compute $G^T \cdot u$, where u is an arbitrary vector. Since every step in computing $G^T \cdot u$ can be treated as an elementary transformation of the matrix, the algorithm is a product of many elementary transformations. So we can compute $G \cdot v$ by considering the transposition of these elementary transformations. And if we can compute $G^T \cdot u$ in $O(T(n))$, we can also compute $G \cdot v$ in $O(T(n))$.

Then we consider the bivariable EGF for $G_{n,k}$, $H(x,z) = \sum G_{n,k} \frac{x^i}{i!}\frac{z^j}{j!}$. We can see $H(x,z) = e^{zODD(x)} = e^{\frac{z}{2}(\ln(1+x)-\ln(1-x))}$. Let $H_i = [x^i]H$, from the differential equations

$$\frac{\partial G}{\partial x} = \frac{z}{2}(\frac{1}{1+x} + \frac{1}{1-x})G = \frac{z}{1-x^2}G$$

we can derive a linear recurrence formula for H_i,

$$iH_i = zH_{i-1} + (i-2)H_{i-2}$$

We can define an 2×2 matrix A_i, and let $[H_i, H_{i-1}]^T = A_i[H_{i-1}, H_{i-2}]^T$. If we want to compute $\sum H_i \times u_i$, you can use the divide and conquer algorithm with FFT in $O(n \log^2 n)$. It's an algorithm for computing $G^t \cdot u$, thus we can compute $G \cdot v$ in $O(n \log^2 n)$ too.

So we solved this problem in $O(n \log M + n \log^2 n)$.

The second solution

We may notice Burnside's lemma is not inherent here. So we can solve this problem without it. There may be many other combinatorial solutions, but I include a generating function based solution here, translated from here.

Let x be a variable in the Set FPS, where multiplication is xor covolution. And t is a variable in the FPS to record how many numbers chosen. The answer is

$$[x^X t^N] \prod_{S=0}^{M} \frac{1}{1 - tx^S} = [x^X t^N] \prod_{S=0}^{M} \frac{1 + tx^S}{1 - t^2}$$

Let $F = \prod_{S=0}^{M} \frac{1+tx^S}{1-t^2}$. We can use FWT, and IFWT to extract the coefficients of $[x^X]$,

$$[x^T]FWT(F) = \prod_{S=0}^{M} \frac{1 + (-1)^{|S\cap T|}t}{1 - t^2}$$

, and

$$[x^X]F = [x^X]IFWT(FWT(F)) = \frac{1}{2^L} \sum_{T=0}^{2^L-1} (-1)^{|X\cap T|} \prod_{S=0}^{M} \frac{1 + (-1)^{|S\cap T|}t}{1 - t^2}$$

We can notice $\prod_{S=0}^{M} \frac{1+(-1)^{|S\cap T|}t}{1-t^2}$ is only related to the number of odd numbers and even numbers of $|S \cap T|$. Let $g(T) = \sum_{S=0}^{M} [2 \mid |S \cap T|]$. So

$$F = \frac{1}{2^L} \sum_{T=0}^{2^L-1} (-1)^{|X\cap T|} (\frac{1+t}{1-t})^{g(T)}$$

.

With some digital dp, or analysis of the Trie structure of $0 \sim M$, we can find see there are at most $O(L)$ different values for $g(T)$, and $\sum (-1)^{|X\cap T|}$ for all T with the same $g(T)$ can also be computed in $O(polylog(M))$. We omit the details here.

So F be become a polynomial of $\frac{1+t}{1-t}$ with $O(\log M)$ non-zero terms. Let $P(t) = (1+t)^a(1-t)^{-a}$, $P'(t) = \left(\frac{a}{1+t} - \frac{a}{1-t}\right) P(t)$, which is P-recursive, so we can compute the first N terms in $O(N)$.

Then this problem can be solved on $O(N \log M)$.

Problem B. Best Problem

First, we change the notations to make the solution easier to understand. For two adjacent bits, if they are `00`, we write down an `R` between them. If they are `11`, we write down an `L`. If they are `01` or `10`, we write down an `X`. WLOG, we add two `1`s to the left of the string, and two `0`s to the right of the string.

Consider the way how `0101` changes:

1. `001010 -> 010100`, which means `RXXXX -> XXXXR`
2. `101011 -> 110101`, which means `XXXXL -> LXXXX`
3. `001011 -> 010101`, which means `RXXXL -> XXXXX`
4. `101010 -> 110100`, which means `XXXXX -> LXXXR`

The new problem is, there is a string, we can move an `R` to 4 cells right, and an `L` to 4 cells left, and the path from the start to the destination must be `X`. If there is an `RL` pair, where `R` is 4 cells left to `L`, they can cancel each other out. For consecutive `X`s, we can produce `LR` pairs from that. By parity, we need to ensure there are odd `X`s between `LR`, and even `X`s between `LL` and `RR` pairs.

We will do operation 3 first. We can cancel out all matched `RL` pairs first greedily. Since the remainder modulo 4 of all positions of `L` and `R` will not change, there will not be useful operation 3 anymore.

Then for two consecutive `R`s, we will move the left `R` to right by the first operations greedily. If we use operations 3 and 4, these newly produced pairs will finally be canceled out. The number of steps will not change. We can do similar things for `L`.

After these greedy steps, the sequence will be turned into something like `L...RR..RR.......L..LL.......RR.........LL.....R.` There are many `LR` groups pointing to each other. For each `L` or `R` group, there will be zero or two `X`s between them.

Then we will consider moving these entire `L` or `R` groups to 4 cells left or 4 cells right until they meet together, and then produce new `LR` pairs by operation 4 at the end.

Finally, we need to do dp on it, to record where each `L` and `R` group will go, and what is the maximum number of steps. The weight is a quadratic function, so it's possible to solve it by convex hull trick. What's more, since we are trying to find the maximum, for some reason, the maximum can be reached on the leftmost or the rightmost position. It means for each pointing pair, only one of `LR` groups will move. Then we can get rid of the convex hull trick and solve it in simply $O(n)$.

Problem C. Cryptography Problem

The first solution

Let $M = \lfloor \frac{p}{200} \rfloor, x' = (a_1 x - c_1 + M) \bmod p$. So we know $x' \in [0, 2M]$. We can rewrite all other equations in terms of x', and we try to solve x' instead of x. I define x to this x' in the following part.

The basic idea of the solution is you need to find some random linear combinations of equations that make the coefficient small and don't amplify the error term.

If we multiply the i-th equations by y_i and add them together, we can get a new equation $(\sum a_i y_i) \times x + \text{error term} \equiv (\sum c_i y_i) \pmod p$. And the error term will not be greater than $(\sum |y_i|) \times M$. Let $A = (\sum a_i y_i) \bmod p$, $E = (\sum |y_i|) \times M$, $C = (\sum c_i y_i) \bmod p$. If the coefficient A is small enough and the error term E doesn't exceed $p/2$, the solution of this equation can be represented by C intervals $[\frac{C+ip-M}{A}, \frac{C+ip+M}{A}]$.

We can intersect these intervals with the old solution set iteratively, and keep a set of intervals that x can be. If we can find the linear combinations randomly, and intersect them with proper random solutions. The set will become smaller exponentially. "Proper solutions"here mean if the interval is a bit smaller, you can try to intersect them with larger A. Otherwise, the set may not become smaller very fast.

The next question is how to find these random linear combinations. This technique is always used in hacking rolling hash, which is called multi-tree-attack. Detail here. The brief idea is that you have a set of numbers, and you can sort them, and generate a new set from the difference of near numbers. For example, in this problem, we can generate a new set $\{a_i - a_j | j < i \le i + 5\}$ from a sorted sequence a, and keep the smallest numbers among them. and do it 5 times. It's good enough to pass this problem.

The second solution

This problem is well known in the CTF community, and thank @oToToT and @toxicpie for teaching me that. This problem can be reduced to the shortest vectors in a lattice problem, and solved by some library codes.

We choose any T clues and build construct $T+2$ vectors $e_1, e_2, \ldots, e_{T+2}$ with size $T+2$.

- For $i = 1, 2, \ldots T$, let $e_{i,i} = M_1 p$.
- For $i = T+1$, let $e_{i,j} = M_1 a_j$ for $j = 1, 2, \ldots, T$ and $e_{i,T+1} = 1$.

- For $i = T + 2$, let $e_{i,j} = M_1 c_j$ for $j = 1, 2, \ldots, T$, and $e_{i,T+2} = p/M_2$.

If the answer is x, $xe_{T+1} - e_{T+2} - \sum_{i=1}^{T} \lfloor a_i x/p \rfloor e_i$ is a short vector with every dimension roughly $\max(p, M_1 \cdot \text{error term})$.

We can select $T = 14, M_1 = 50, M_2 = 20$ to pass this problem, but you need a well-optimized LLL library code.

Problem D. Digit Sum Problem

The main idea of this problem is something like sqrt decomposition or meet in the middle.

Let $M = \sqrt{n}$, and $M_2 = 2^p, M_3 = 3^q$ be the nearest power to M. If we listed all multiples of M_2 and M_3, we can divide the interval $1 \sim n$ to $O(\sqrt{n})$ pieces. The question is how to compute the sum for each interval in $O(1)$ with some precalculations.

For each piece, the prefix of numbers in binary and ternary is fixed. So we don't need to consider the prefix. And two endpoints of this interval should be a multiple of M_2 or M_3. So there are at most $O(M_2 + M_3)$ different states. If one end is the multiple of M_2 and the other end is the multiple of M_3, the state is determined by its length. If both ends are multiple of M_2, the state is determined by the remainder modulo M_3 of the first number. It's similar to M_3.

The next question is how to compute answers for these states. We can compute the answer for $(2M_2, M_3)$ or $(M_2, 3M_3)$ from $O(M_2, M_3)$ in $O(M_2 + M_3)$. For example, for all different states of $(2M_2, M_3)$, we can divide it into $O(1)$ intervals by the multiples of M_2 and M_3, and sum them up. So we can start from $(1, 1)$, and increase the smaller power until they both reach $O(\sqrt{n})$.

The time complexity is $O(\sqrt{n})$.

Problem E. Elliptic Curve Problem

First we notice that the condition $[l \le (x \bmod p) \le r] = \lfloor \frac{x-l}{p} \rfloor - \lfloor \frac{x-r-1}{p} \rfloor$. And $1^2, 2^2, \ldots \left(\frac{p-1}{2}\right)^2$ are all different quadratic residues of p.

We can rewrite the problem to $\sum_{i=1}^{(p-1)/2} [l \le (i^2 \bmod p) \le r] = \sum \left(\lfloor \frac{i^2-l}{p} \rfloor - \lfloor \frac{i^2-r-1}{p} \rfloor \right)$.

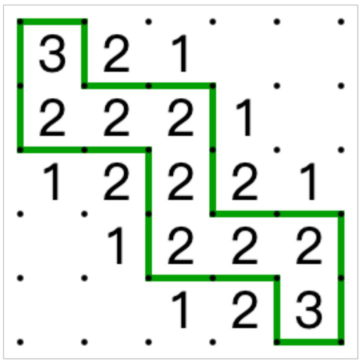
Then we need to compute $\sum \lfloor \frac{i^2+c}{p} \rfloor$ effectively. The technique is quite similar to computing $\sum \lfloor \frac{n}{i} \rfloor$.

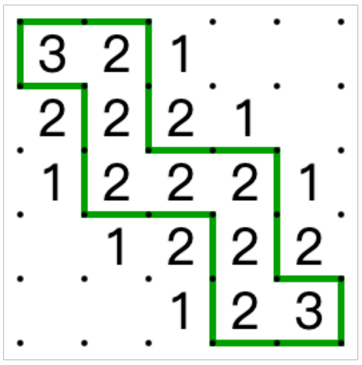
The basic idea is we need to compute the number of integral points between this parabola and the positive x-axis. The region is convex, and the range of coordinates of points are in $[0, p + O(1)]$. So the number of edges of the convex hull is $O(p^{2/3})$. We can find the convex hull and then compute the answer.

We don't include the details of this technique here. To who is interested, there is a short survey(in Chinese) about this technique.

Problem F. Full Clue Problem

The answer:





You may find it by writing a brute force on small n or just playing by hand.

Problem G. Graph Problem

Let A be the adjacent matrix of the graph. The reachability matrix can be something like that $I + A + A^2 + \cdots = (I - A)^{-1}$. We don't consider the convergence here. In our solution, we can assign a random weight to each edge (denote this new weighted adjacent matrix by W). Then we can compute the inverse matrix $N = I - W, M = (I - W)^{-1}$. s can reach t iff $M_{i,j} \neq 0$.

If we delete several vertices of the graph, we will apply a low-rank update on the adjacent matrix, and we need to know the value of some specific elements of this new inverse matrix, which can be done by Woodbury matrix identity:

For an $n \times k$ matrix U and $k \times n$ matrix V, $B = A + UV$, then $B^{-1} = A^{-1} - A^{-1}U(I_k + VA^{-1}U)^{-1}VA^{-1}$.

Let $U_{i,j} = [i = p_j], V_{j,i} = W_{p_j,i}$. $(UV)_{i,j} = \sum_k [i = p_k] W_{p_k,j}$, so $(UV)_{i,j}$ is $W_{i,j}$ if i is in p. $N + UV$ is the matrix that removes all outgoing edges for vertices in p.

Then we are going to compute the intermediate matrix $I_k + VMU$.

$$(VMU)_{i,j} = \sum V_{i,k_1} M_{k_1,k_2} U_{k_2,j} = \sum W_{p_i,k_1} M_{k_1,p_j} = \sum (I_{p_i,k} - N_{p_i,k}) M_{k,p_j}$$

$\sum N_{p_i,k} M_{k,p_j} = [i = j]$ by the definition of the inverse matrix. So $(I + VMU)_{i,j}$ can be simplified to M_{p_i,p_j}.

$(MU)_{s,i} = \sum_k M_{s,k} U_{k,i} = M_{s,p_i}$. $(VM)_{j,t} = \sum_k V_{j,k} M_{k,t} = \sum_k (I - N)_{p_j,k} M_{k,t} = M_{p_j,t} - [p_j = t]$. But p_j doesn't equal t, so we can simplify this term to $M_{p_j,t}$.

In conclusion, let $P_{i,j} = M_{p_i,p_j}$, s can reach t iff $M_{s,t} \neq \sum M_{s,p_i} P^{-1}_{i,j} M_{p_j,t}$.

For each query, we need to compute the inverse matrix of this intermediate matrix in $O(k_1^3)$, and answer each query in $O(k_1^2)$. The time complexity is $O\left(n^3 + qk_1^3 + (\sum k_2) k_1^2\right)$.

Problem H. Hard Problem

There are several ways to solve this problem.

The first approach is by the dual of linear programming.

There are several ways to choose intervals (including gapped intervals), denote them be $I_1, I_2, \ldots, I_m$. Let x_i be the number of operations on interval I_i. The constraints is that for every $i (1 \leq i \leq n)$, $\sum_{i \in I_j} x_j = a_i$. And we want to minimize $\sum x_j$. The solution to this linear programming is integral, you can prove it by yourself.

The dual of this problem is that $\sum_{j \in I_i} y_j \leq 1$, and we want to maximize $\sum_{i=1}^{n} y_j a_j$. y_j can be negative here. The solution to this linear programming is also integral, you can prove it by yourself.

The combinatorial meaning is that you need to assign a weight to each position and the sum of weights of all intervals should be not greater than 1.

First we can notice that $y_i \in \{-1, 0, 1\}$. If $y_i < -1$, we change it to -1. Since the sum of every interval is not greater than 1, so the sum of an interval containing y_i is not greater than $1 + (-1) + 1 = 1$.

Let $dp_{i,j,k,l}$ be the maximum sum of $a_i y_i$ if we filled the first i elements, the maximum suffix sum of consecutive interval/odd gapped interval/even gapped interval to i. And we can enumerate the value y_{i+1} and update these three sums.

The time complexity is $O(n)$.

The second approach is by greedy. Translated from here.

You need to choose intervals and gapped intervals and cover the position i exactly a_i times for all $i (1 \le i \le n)$.

Let's start with the first two positions a_1, a_2:

- $a_1 > 0, a_2 > 0$, we will start $\min(a_1, a_2)$ intervals here.
- $a_1 > 0, a_2 = 0$, we will start a_1 gapped intervals here.
- $a_1 = 0, a_2 > 0$, we will skip the first position.

When we consider the i-th position. There are z unclosed intervals and b unclosed gapped intervals that can reach i.

If $z + b \le a_i$, we can subtract $z + b$ from a_i and extend these intervals.

If $z + b > a_i$, we need to close $z + b - a_i$ intervals. But we don't know which intervals we need to close, it may depend on later positions. Let $k = z + b - a_i$, we can close k intervals and k gapped intervals (it may be impossible to close k intervals, we will see how to deal with this case later), then we can start k intervals in both types without cost from i-th position. So we can add k to a_i and subtract k from the answer. Since we will always start k more intervals from a_i but they are free.

Another question is when z or b is smaller than k, it may be impossible to close k intervals. If $z < k$, that is $z < z + b - a_i$ which means $a_i < b$, then we always need to close $b - a_i$ gapped intervals, $k' = z$ now. It's not hard to deal with this case. And it's similar for $b < k$.

Then we solve this problem in $O(n)$. The order of these greedy steps may be a bit tricky since we need to ensure when we do greedy stuff for a_i, a_{i+1}, we need to subtract the gapped intervals that can reach a_{i+1}, and we need to ensure that extra k added to a_i will not be taken by previous intervals. So you may refer to the code of the original article for clarification.

Problem I. Interval Problem

WLOG, we assume $r_1 < r_2 < \cdots < r_n$, and the graph is connected. If the graph is not connected, we can simply compute the answer for each connected component.

First, we consider how to compute the distance from the interval j to the interval $i (j < i)$. We can start from the interval j and jump to the interval that intersects with j and with the rightmost endpoint greedily until it reaches i.

We can notice a tree structure here. For an interval i, let its parent be the rightmost interval that intersects with it.

Then we consider how to compute the distance from i to all intervals j such that $r_j < r_i$. It's not hard to see, for all intervals intersecting with i, the distance is 1, the distance of their children is 2, and so on. So we can compute the answer by something like tree dp and cumulative sum in $O(n)$.

We can compute the distance from i to all intervals j such that $l_j > l_i$ in the same manner. For each interval such that $l_j < l_i$ and $r_j > r_i$, the answer always equals to 1.

So we can solve this problem in $O(n \log n)$.

Problem J. Junk Problem

If there are no bidirectional edges, the problem can be solved simply by LGV lemma. Let $M_{i,j}$ be the number of paths from $(1, a_i)$ to (n, b_j), we can compute the determinant of this matrix.

For the general graph, we can deal with it by Talaska's formula. You'd better read this paper for a better understanding.

The brief idea is we can add two directed edges with weight x for bidirectional edges. The number of paths between two vertices should be replaced by the sum of the weight of all paths. There can be infinite paths. For example, we can walk on a bidirectional edge indefinitely. the sum equals $1 + x^2 + x^4 + \cdots = \frac{1}{1-x^2}$. So the sum of all paths is actually a formal power series. We can see later that we will compute the value when x equals some specific numbers, so we can still compute the sum by simple dp.

From Talaska's formula, the determinant of this new matrix is equal to the collections of nonintersecting and non-self-intersecting paths and cycles, divided by a sum over collections of nonintersecting cycles. To be specific, $\det(M) \times (1 - x^2)^k$ equals

$$\sum_{\text{all disjoint simple paths}} x^{\text{\#bidirectional edge passed}}(1 - x^2)^{\text{\#bidirectional edge unvisited}}$$

It's a polynomial with a degree not greater than $2k$. Let $x = 1$, we can find the answer. But we can not directly substitute x by 1, since the denominator can be 0 during the computation. We can substitute x by $2, 3, \ldots, 2k + 2$ and get the polynomial by interpolation.

The time complexity is $O(n^3k)$.

Problem K. Knapsack Problem

This problem is solvable by solving the linear programming, then trying to enumerate how to round the solution, or just copying an ILP solver, or some heuristics. But they are not intended.

The intended solution is digital dp. A similar problem here.

Basically, the problem is a four-dimension knapsack problem. But if we solve this four-dimension knapsack problem directly, the time complexity will be at least $O(W^4)$, which is not acceptable.

So we can consider fixing the value for each x_i from the highest bit to the lowest bit in binary, and record the remaining volume for these four dimensions. We can see if the volume for one dimension is greater than 15 (the actual volume is $15 \times 2^b + a_i \bmod 2^b$) before we consider the b-th bit, it can not be filled even if we set $x_i = 2^b - 1$ for all remaining bits. So we can find a solution that runs in $O(\log W \times 16^4)$. But it may be hard to fit in the time limit for its large time constant.

We can find the upper bound 15 is not tight. For example, if the numbers of subsets $1, 1234, 12, 134$ are greater than 0, it's always to decrease and increase the pairs $(1, 1234), (12, 134)$ and make the answer not smaller. So there are at most 5 sets used in the optimal solution when we consider one bit. The upper bound of remaining volumes can be reduced to 9, which is enough to fit in the time limit.

Actually, we can show the upper bound can be improved to 7 with ~~a subtle analysis~~ stress test.

Problem L. Linear Congruential Generator Problem

The first solution

First, we can know the result of RNG mod$i$ after i-th execution from the permutation.

Since it's a linear congruential generator, we can get some equations $((a_i \times seed + b_i) \bmod p) \bmod i = c_i$.

We try to solve this from two equations. For example, we choose the first equation M_1 (we can see later, let $M_1 = n/2$ is good enough), let $x = (a_{M_1} \times seed + b_{M_1}) \bmod p$. From $x \bmod M_1 = c_{M_1}$, we can get $x = M_1 t + c_{M_1}$. Here $t \leq p/M_1$, which is roughly in $O(p/n)$.

Then we choose another equations $M = M_1 + M_2$, we have $((a_{M_2} \times x + b_{M_2}) \bmod p) \bmod M = c_M$.

We can substitute x, and get $((a_{M_2}(M_1t + c_{M_1}) + b_{M_2}) \bmod p) \bmod M = c_M$, denoted by $((k_1t + k_2) \bmod p) \bmod M = c_M$, which is a linear congruent equations of t.

We can carefully choose a_{M_2} to make $k_1 = a_{M_2}M_1 \bmod p$ as small as possible, which is roughly in $O(p/n)$. Then $k_1t + k_2$ is in $O(p^2/n^2)$. Let $q = \lfloor (k_1t + k_2)/p \rfloor$, then q is in $O(p/n^2)$, and $(k_1t + k_2) \bmod p = k_1t + k_2 - qp$.

We can enumerate q, then solve the equations for $(k_1t + k_2 - qp) \bmod M = c_M$. We can solve t based on the range of t where $qp \le k_1t + k_2 < (q + 1)p$ and the linear congruent equation. There are about $O(p/k_1/M) = O(1)$ solutions for each q. So we can check all possibilities by brute force. If t is not correct, it will reject in $O(1)$ time.

The time complexity is $O(p/n^2 + n)$.

The second solution

Thank @toxicpie for teaching me that.

We have a sequence of equations like $((a_ix + b_i) \bmod p) \bmod i = p_i$.

Take all the i such that $i \bmod 1000 = 0$, so we get 100 equations like $((a_ix + b_i) \bmod p) \bmod 1000 = c_i$, so $(a_ix + b_i) \bmod p = 1000k_i + c_i$ for some $0 \le k_i \le p/1000$ and then $(a_i/1000) \times x - k_i \equiv (c_i - b_i)/1000 \pmod p$, which is just problem C.

Problem M. Minimum Element Problem

First, we can see two permutations are equivalent iff their Cartesian trees are the same. If no element is fixed, the answer is simply $C_n = \frac{1}{n+1}\binom{2n}{n}$. Since there is at least one permutation for each binary tree with n vertices.

If $p_x = y$, there are two cases becoming invalid.

- The depth of x is greater than y, since x should be greater than all ancestors.
- The size of the subtree of x is greater than $n + 1 - y$, since x should be less than all descendant.

And these two cases are independent. If x violates both conditions, the number of vertices will be greater than n.

So we solve these two cases separately. We compute the ways that the depth of x equals y for $y = 1 \sim n$, and the size of the subtree of x equals y for $y = 1 \sim n$.

For depth, we consider the path from x to the root. If there are p vertices on the left of x(denoted by $l_1 < l_2 < \cdots < l_p$) and q vertices on the right of x (denoted by $r_1 > r_2 > \cdots > r_q$). Let $l_0 = 0, l_{p+1} = x$, $r_0 = n + 1, r_{q+1} = y$. The number of ways is

$$\prod_{i=1}^{p} C_{l_i - l_{i-1} - 1} \times \prod_{i=1}^{q} C_{r_{i-1} - r_i - 1} \times \binom{p+q}{p}$$

The elements from $l_{i-1} + 1$ to $l_i - 1$ form a binary tree independently, so the number of ways is $C_{l_i - l_{i-1} - 1}$. The right part is similar. There are $\binom{p+q}{p}$ ways to interleave l and r to form the path from x to the root.

For a fixed p, there is a closed formula (denoted by L_p) for $\sum_l \prod_{i=1}^{p} C_{l_i - l_{i-1} - 1}$, called k-th convolution of Catalan, which is basically something like $\frac{k+1}{n+k+1}\binom{2n+k}{n}$. Details here.

So if the depth of x is y, the number of ways is $\sum_{p+q=y-1} L_p \times R_q \times \binom{p+q}{p}$. It's a simple convolution.

For subtree, we can consider shrinking all vertices in the subtree to a single vertex. And this vertex is a leaf.

The subproblem is to compute the number of binary trees with n vertices and k-th vertex is a leaf. Surprisingly, the answer is C_{n-1}, unrelated to k. It is not hard to see we can build a bijection by removing this leaf in a binary tree with n vertices, and adding a leaf in a specific position of a binary tree with $n-1$ vertices.

So if the size of the left subtree of x is p and the size of the right subtree is q, the number of ways is $C_p \times C_q \times C_{n-p-q-1}$. It's also a simple convolution.

Then we solve the problem in $O(n \log n)$.