Problem A. P=NP

Note that this is much easier than the actual P=NP problem...

So the constraints should be small enough you can loop over all P where $0 \leq P \leq X$ and all N where $0 \leq N \leq Y$.

Alternatively you can notice there is only a solution when either $P = 0$ or $N = 1$. There are $Y + 1$ pairs where $P = 0$ $(\{0,0\},\{0,1\},\{0,2\},\cdots\{0,Y\})$ and there are also $X + 1$ pairs where $N = 0$, $(\{0,1\},\{1,1\},\{2,1\},\cdots\{X,1\})$. This counts the pair $0,1$ twice, so we have to subtract 1 at the end. The end answer is $(X+1)+(Y+1)-1$ or just $X+Y+1$. Complexity $O(1)$.

Problem B. Average

The average of a set of numbers is just the sum of all the elements divided by the size of that set. To maximize the average we maximize the sum of the remaining elements because the amount of elements will always be $N-1$ after we remove one element. Therefore it is always optimal to remove the smallest element. To get the average rounded to the nearest tenth, multiply everything by 10 when summing or getting the minimum to keep everything an integer. Then at the end divide it by ten and convert it to a double to get the answer. Also note that it is always optimal to remove an element, as the smallest element of the array is $\leq$ the average. Finding the sum and the minimum of an array is just linear time, so the complexity is $O(N)$.

Problem C. Lattice Flowers

It will always to optimal to draw a straight line along one of the edges from point $(1,1)$. So you will go to either $(1,m)$ or $(n,1)$. So the first number will always be $\max(n,m)$. The second number will be 1 if $n \neq m$ and 3 if $n = m$, 1 since there will only be 1 longest edge and 3 since you can draw from $(1,m),(n,1),or(n,m)$ (note that since $n = m$ here we can use the two interchangably). Complexity $O(1)$ per testcase.

Problem D. Abc's (Easy Version)

To see which letter would win a point on a certain turn, we need to keep a tally of how many times each letter has been played by the computer. However this is not the only thing we have to track of. For each turn we also need to keep a running tally of how many points the person will get with each card if that was the only card they played, and at the end we output the highest value of the three. We only have to interate through once to get the necesarry information so the complexity is $O(N)$.

Problem E. Calculating Costs

We need to find the mean, median, and mode of the list of numbers. The mean is the average of all the numbers, meaning that we need to find the sum of the list of numbers, then divide it by the number of numbers in the list. To find the median, we first have to sort the array, and if there is an odd number of numbers in the list, then we find the middle number, and if it there is an even number of numbers in the list, then we find the average of the two middle numbers. To find the mode, we count how much each number occurred in the list, then find the number that occurred the most. After we find the mean, median, and mode, we print them out, all in separate lines.

Total complexity, $O(n \log n)$ or $O(n)$ depending on the implemenation of the sorting algorithm.

Problem F. Books

The algorithm to this problem is pretty simple. All you need to do is to first sort the list of books by alphabetical order first. Then, within the groups of books with the same first letter, you sort them by the year they were published, from the oldest to most recent.

Problem G. Matching Mispronunciations

For this problem, we need to have a way of checking the "difference"between two words. We do this by traversing through each letter and seeing if they're the same or not, and seeing how many letters are the same, and how many aren't. If there is exactly one letter pair that is off, then we know that that word is the word we're looking for. After we traverse through each given word, checking for the difference, we can find the correct words that we need for the order, then we print that out.

Problem H. Position of Set

Note that the only time there are multiple different ways to order a problem set is when a difficulty d appears multiple times. If the frequency of a difficulty d in a list is c_d, then the number of ways that the problem with difficulty d can be ordered is $c_d!$ since there are c_d different indicies to be distrubited. The answer is then just the product of $c_d!$ over all d. We can precompute the factorials and frequencies in $O(N)$ time, and thus the complexity is linear.

Problem I. Chessbot's Lawn

Let $pre(i, j)$ be the answer to the query $x = i$ and $b = j$. Notice that $pre(i, j)$ is just the minimum of the minimum value in the reactangle from $(0, 0)$ to $(i, j-1)$, the minimum value in the rectangle from $(0, 0)$ to $(i-1, j)$, and the value at (i, j) itself. We can write this as $pre(i, j) = min(pre(i-1, j), pre(i, j-1), A[i][j])$ where $A[i][j]$ is the beauty value at square (i, j). Each of the $N \cdot M$ states has constant trasition time, so the total complexity is $O(N \cdot M)$.

Problem J. Making Stonks

The solution to this problem is to binary search over the final answer time, then run a function to check whether or not each answer is possible. To find if a time v works as the answer or not, we can loop through all the people and sum how much money each of them made by time v. The amount of money the ith person makes is $(v - t_i)/r_i$, where t_i is the time the person is hired and r_i is the frequency he makes money. However, this value is negative when v is before the person is hired, so it is important that we take 0 insted if the value is negative. If the total sum is $\geq x$, then the v works and the upper bound of the binary search is set at v. One thing to care about in this problem is possibly overflowing the long long limit because the sum for each query can be as large as $n * v$. n can be 10^5, and v goes up to 10^{15}, so $n * v$ might go to 10^{20}, above long long limit. The solution to this issue is to check after each person's earned amount if the target has been hit, and if so just immediately stop.

Problem K. Necklaces

One solution is to use sets. Keep a set containing where each string begins, and a set containing where each string ends. When quering for a node, simply find the closest start point and end point use lower/upperbound. Also note we store the beginning indices as negative values to query for highest index lower than a value in the set. Because updating and querying on a set is $\log N$ the total complexity is $N(\log N)$.

You can actually get linear by using disjoint set, answering the queries offline by adding edges instead of substract them. Then you store the beginning and ending index of the component each node is in. You can then get $O(N \cdot inv\_ack)$ which is essential linear.

Problem L. Cooked Fish (Easy Version)

So lets say (a, b) is a valid answer. Let's define n as $\sum_{i=a}^{b-1} i$ and try to rewrite n.

$$n = \sum_{i=a}^{b-1} i = \sum_{i=0}^{(b-a)-1} a + i = (b - a - 1) \cdot a + \sum_{i=0}^{b-a-1} i = (b - a) * a + (\frac{(b - a - 1) \cdot (b - a)}{2})$$

So i started out with the definition of n, then I subtracted a from all i, then i removed a from the sigma since it's independant, then finally I simplified the sum of the first $b-a$ nonnegative integers with gauss's. So now, if (a, b) is a valid solution, we can express a in terms of n and $b-a-1$. Let's define k as $(b-a)$ (note this is different from k in the problem statement). We have

$$n = k * a + \frac{k \cdot (k-1)}{2}.$$

If we set $a = 0$, once k grows large enough, it's side of the equation will be greater than n. namely when k is around $\sqrt{2} \cdot n$, $\frac{k \cdot (k-1)}{2}$ will be around the size of n. So as result there are only around $O(\sqrt{n})$ values for k that are suitable. So now let's rewrite our equation.

$$\frac{n - \frac{k \cdot (k-1)}{2}}{k} = a$$

So since since we know that a is an integer, $(n - \frac{k \cdot (k-1)}{2})$ must be divisible by k for a to end up as an integer. So if we have a, $b = a + k$. So we can check all $O(\sqrt{n})$ values of k and check if there is a solution. Also note by some math magic, there is no solution for the problem when n is a power of 2. The end complexity is $O(\sqrt{n})$.

Bonus: prove that there is no solution for when n is a power of 2, also trying solving this for $n \leq 10^{18}, k = 1$.

Problem M. Shion's Feast

We want the lcm to be as large as possible. The lcm is large if we have two large numbers that are relatively prime since their lcm is their product. Motivated by this, we ask: is there a large number we can add that will be relatively prime to the maximum number in the array?

Let MX be the maximum number in the array. Realize that MX and $K \cdot MX - 1$ are relatively prime ($gcd(MX, K \cdot MX - 1) = 1$) using the Euclidean algorithm. Here's an intuitive (and equivalent) proof. Let d be a divisor of MX, then d also divides $K \cdot MX$. Thus, the only way that d would divide $K \cdot MX - 1$ is if it divides -1, so d must be 1 if it does).

Therefore, we add $B = K \cdot MX - 1$ if MX is not 1 otherwise we add $B = K \cdot MX$.

Time complexity: $O(N)$

Problem N. Teleport

Note that the graph of teleports makes a directed acyclic graph with 1 or more components each having one endpoint which all nodes in that component lead to. Then all we have to do is go through all the teleporter start points and assign the final position it goes to. Save the final end point each teleporter goes to, which makes it so that you can find the endpoints of all teleporters in amortized $O(K)$ time. Bfs can then be used to finish the problem. the complexity is $O(N^2)$.

Problem O. Kanna's Field of Flowers

The answer is clearly at least N since we can just select any row. Call the original grid A. Let's create a new $N - 1 \times N$ grid B where $B[i][j] = 1$ if $A[i][j] == A[i+1][j]$. Realize now that any $W \times L$ rectangle in A where all rows are the same is just a $W - 1 \times L$ rectangle of 1's in B. The problem has been simplified to finding the largest rectangle of all 1's in B.

This is a well known problem which can be solved by iterating over the bottom row of the rectangle. Now for each bottom row, we find the maximum area of a rectangle in the histogram of 1's with base at that bottom row (the bars of the histogram are consecutive 1's in a column where the last 1 is at the bottom row). To do this, realize the top of the largest rectangle must be at the top of one of the bars of the histogram. We iterate over this top, so we have already bound the top and bottom of the rectangle and only need to bound the sides. To do this, we just need to find the first bar of the histogram to the left and right of the current bar that has a smaller height/top. We can find this for all bars in $O(N)$ using

NSE with a stack. It takes $O(N)$ time for each bottom row and thus $O(N^2)$ time in total. Let C be the converted area of that rectangle in A. The answer is $max(N, C)$.

Time Complexity: $O(N^2)$

Problem P. Abc's (Hard Version)

We will solve the problem using dp. Let $dp(i, k)$ be the answer when i is the amount of turns, and k is the last letter played. So find this value we have two cases.

Case 1: $i \neq N$ We have $dp(i, k) = max(dp(j, k) + pre(i, k) - pre(j, k))$, where $k < j \leq i - k$ and $pre(a, k)$ is the amount of points the player will get if you played card a for the first k turns

Case 2: $i = N$ We have $dp(i, k) = max(dp(j, k) + pre(i, k) - pre(j, k))$, but this time $k < j \leq i$ because at the end there doesn't need to be K cards of the same type.

To do this in linear time, define $dp_max(i)$ as the max of $dp(i, k)$ across all k. We then maintain an array val where $val[k]$ is computed as $max(dp_max(i) - pre(j, k))$ for each k, and have $dp(i, k) = val[j] + pre(i, k)$.

With this adjustment, it is solvable in linear time.

Problem Q. Tree Width

Note: I probably overestimated the difficulty of this problem. I made a solution that ran in $O(n \cdot tree\ depth)$ that might have used linear memory if i changed it. Most submissions ran in $O(n \cdot tree\ depth)$ memory as well as time. I didn't realize it was so much easier to implement using a bottom up method if you could store the entire dp table without mle (which you can).

The first important observation is that the height of a modtree is rather small, you could have arrived at this conclusion from guessing, seeing that it's close to a random tree and random tree's have low depths, or making the assumption that the height of the modtree is log2 since mod is half. By running a $5e5^2$ brute force you can find that the largest depth a modtree goes up to is 42 with $n = 493919$.

Let's define $cnt[i][j]$ as the amount of nodes with distance j away from node i. Node i has distance 0 from itself.

Now let's define $temp[i][j]$ as the amount of nodes in node i's subtree. (Note we root the tree at node 0).

$par[i]$ is the parent of node i, parent of the root is undefined.

$dep[i]$ is the depth of node i, $dep[0] = 0$.

Then we have

$$cnt[i][j] = cnt[par[i]][j-1] + temp[u][j] - temp[u][j-2]$$

Note that in this equation if ever you access a negative index, it is just 0.

So in words, the amount of nodes distance j away from node i is

1. the amount of nodes distance $j - 1$ from the parent of i. This is the amount of nodes up the tree that contribute to the answer. Note this overcounts since there are also some nodes with distance $j - 1$ from $par[i]$ within $i's$ subtree. We will handle the overcount in case 3 2. the amount of nodes distance j from node i in $i's$ subtree. This is the contribution down the tree on in $i's$ subtree. 3. subtract the amount of of nodes in $i's$ subtree distance $j - 2$ from node i. These are the nodes that are overcounted in case 1. Since these nodes have distance $j - 1$ from $par[i]$ but do not have distance j from node i.

The answer is then $\max(cnt[i][j])$ over all i and all j.

I think most teams made the above observations, but our implementations are different.

So let's say I find $cnt[0][j]$ for all j. Then I want to transition from each parent to their child. We can just compute the $temp$ array for the child and then loop over all relevant j and recompute $cnt[i][j]$ and take the max. The I repeat the process from the children. The total complexity runs in $O(\sum_{i=0}^{n-1} subtree\ size\ of\ i)$. This evaluates to $O(n \cdot tree\ depth)$. Consider the sum of all subtrees of all nodes with a fixed depth from

the root. The sum of all of these subtrees is at most n since each node can only belong to one subtree of a node of a certain depth. So the sum of all subtrees is around $O(n \cdot tree\ depth)$.

In my implementation, I don't actually store all *cnt* and *temp* values. Instead whenever I recurse, I first store the values in the *cnt* array, but then I edit the array and restore it at the end. You can find my implementation on github /treewidth/model.cpp

Problem R. Kakyoin's Painting

It makes sense to iterate over the number of rectangles in the final painting to make our life easier. How many paintings have X rectangles? Realize that in each column and row, if there are 1's they will belong to only 1 rectangle. This means that the set of columns and rows that have a 1 is a property of every painting.

Let's go from this set to the painting then. If we select X disjoint intervals (for example 2 disjoint intervals could be $[1, 3]$ and $[4, 5]$) for the rows and columns separately, this can uniquely determine $X!$ rectangles. The $X!$ rectangles come from a permutation P. If we fill in 1's in the intersection of the ith interval for the rows and P_ith interval for the columns, we end up with a unique painting.

UwU

Let S_i be the number of distinct ways to choose i disjoint intervals. Then the answer to the problem is:

$$\sum_i i \cdot {S_i}^2 \cdot i!$$

What is S_i? Using stars and bars, it's just:

$$\binom{N+i}{2i}$$

The stars are the indexes of the row/column. The i intervals need to have at least width 1, so we subtract i stars. There are $2i + 1$ sections (intervals or gaps between intervals) so we insert $2i$ bars and choose $2i$.

By precomputing the mod of factorials and inverse factorials, we can find the sum for each i in $O(1)$ time.

UwU

Time complexity: $O(N)$

Problem S. Cooked Fish (Hard Version)

Note this is for the original problem and not for the case of minimizing a in the answer. rip my checker....

This problem is spilt into three main cases. $k \geq 3$, $k = 2$, and $k = 1$.

Let's try solving it for $k = 3$. We can see that once we take a value greater than 10^6 or $10^{18\frac{1}{3}}$, 10^{6k} will exceed n. Then if we define $arr = \{1, 2^3, 3^3, \cdots, 10^{6^3}\}$, then the problem is reduced to finding a subarray with a sum of n. Then we can run an $O(n^{\frac{1}{3}})$ two pointers brute force to find the answer. For all $k \geq 3$, we can solve it in $O(n^{\frac{1}{k}})$ time.

If n is not a power of 2 , there is a solution. Let's prove it. So for a valid solution (a, b), lets define $k = b - a$ (like the easy version). Again, k here does not refer to the k in the problemstatement but rather $b - a$. So we have

$$n = k \cdot a + \frac{k \cdot (k+1)}{2}$$

and

$$2n = 2 \cdot k \cdot a + k \cdot (k+1) = k \cdot (2a + k + 1)$$

So we know that $2n$ has an odd and an even factor since the parity of k and $(2a + k + 1)$ are different. So if n is a power of 2, then there are not two such factors. Also if we can find any two valid factors, they

make a solution since we can use it solve for a and b. So now if have some n which is not a power of two, we can trivially find an even factor, lets call it y as the largest power of 2 that divides $2n$. Then we define $z = (2n)/y$. So we know that $|z - y|$ is $2 * a + 1$ and that $\min(y, z)$ is k. Then we can solve for a and k and output $(a, a + k)$ as our answer. Complexity $O(1)$ or $O(\log n)$ depending on the implementation.

Let's use our idea for the easy version here. So for a valid answer (a, b) we have $n = \sum_{i=a}^{b-1} i^2$. If we define the sum of integers from $[0, i]$ as $sum(i)$ and the sum of squares from $[0, i]$ as $sumsq(i)$, we can start simplifying

$$n = \sum_{i=a}^{b-1} i^2 = \sum_{i=0}^{b-a-1} (a+i)^2 = \sum_{i=0}^{b-a-1} a^2 + 2ai + i^2 = (b-a) \cdot a^2 + \sum_{i=0}^{b-a-1} 2ai + i^2$$

$$= (b-a) \cdot a^2 + 2 \cdot a \cdot sum(b-a-1) + sumsq(b-a-1)$$

Which again is a sum that can make a only relative to n and $b - a$. For the $(b - a) \cdot a^2$ term, i just took it out of the sigma, for the $2 \cdot a \cdot sum(b - a - 1)$ i used distributive property on the sum of integers from $[0, b - a - 1]$, and i just wrote $sumsq(b - a - 1)$ since that's what the sum of all the i^2 was.

Now believe it or not, we can make a quadratic such that when give n and $b - a$, we can solve for a.

$$(b-a) \cdot a^2 + (2 \cdot sum(b-a-1)) \cdot a + (sumsq(b-a-1) - n) = 0$$

Now we can plug it into the quadratic formula and use $(b - a)$ and n to solve for a.

$$a = \frac{-(2 \cdot sum(b-a-1) \pm \sqrt{(2 \cdot sum(b-a-1)^2 - (4)(b-a)(sumsq(b-a-1) - n)))}}{2 \cdot (b-a)}$$

Now when $sumsq(b - a - 1) - n > 0$, the square root of the discriminator will simplify to be less than $2 \cdot sum(b - a - 1)$ and the entire equation will be negative. Since $sumsq$ grows cubicly, we only have to check around $n^{\frac{1}{3}}$ values of $b - a$. But this is not all, evaluating $sqrt$ takes $O(\log n)$ time. So we have a complexity of $O(n^{\frac{1}{3}} \log n)$ which happens to be too slow.

I have no proof of the following heurstic, but it somehow speeds up the code by around 20 times. So now as we loop over $b - a$, evaluating $sqrt$ is the most time consuming part. But you can notice that if the value in the $sqrt$ is not divisible by i^2, there is no chance that it'll simply to a being an integer. So the optimization is to simply not eval $sqrt$ and continuing if the discrim for a certain $b - a$ is not divislbe by i^2. The end complexity is at most $O(n^{\frac{1}{3}} \log n)$, but I can't prove what it actually is.

I really did not think that the grader was wrong for this :<. My model solution is ac but some code that only handles the case when $k = 1$ is also ac. In fact, harry submitted code that just cout's -1 and its ac. sorry and i hope it was still a good problem :<.

Problem T. Vivy's Singularity Project

Consider a graph where there is an edge $i \to j$ if j is a child of i. The singularity project is successful if the final graph of events is a line. There are an exponential number of ways we can create this line, so we are motivated to use dynamic programming.

When an event occurs, it must be the child of one of the endpoints for the graph to remain a line. This means we only need to keep track of the endpoints in our dp. Let $DP[i][j]$ be the probability that we create a line graph with events i and j as endpoints after i events have occured. Let $P(j)$ be the probability that the ith event is a child of event j. Then, $P(j) = (\lfloor \frac{X}{i} \rfloor + (j < X \mod i))/X$. Our base case is $DP[1][0] = 1$. The transitions are as follows:

$$DP[i][i-1] = \sum_{j=0}^{i-2} P(j) \cdot DP[i-1][j]$$

.

$$DP[i][j] = P(i-1) \cdot DP[i-1][j],\ 0 \le j < i-1$$

.

Thus, we already have an $O(N^2)$ solution to the problem.

How can we do better? Notice that for the first type transition to compute $DP[i][i-1]$, $P(j)$ only takes on at most 2 values. From $j = 0 \ldots (X \mod i) - 1$, $P(j)$ is $\lfloor \frac{X}{i} \rfloor + 1$ and from $(X \mod i) \ldots i-2$, $P(j)$ is $\lfloor \frac{X}{i} \rfloor$. Thus, the j that have the same value of $P(j)$ are contiguous. We simply need to find $\sum DP[i-1][j]$ for the 2 ranges. Notice that for the second type of transition, we are simply multiplying each value in the range $DPi-1][0 \ldots i-2]$ by $P(i-1)$.

Does this sound familiar? Well, we can use a segment tree with lazy propagation to handle these range sum and range multiply queries. In the ith iteration, the array that the segment tree represents will be $DP[i]$. Updating the segment tree to represent $DP[i+1]$ from $DP[i]$ requires only 4 queries each of which take $log(n)$ time, so the time complexity is $O(N \cdot log(N))$.

Time complexity: $O(N \cdot log(N))$.

Problem U. Rengoku's Flame Breathing

The first thing to notice is that the expected number of attacks to reduce the health of a heart to 0 is independent of other hearts. Thus, we can find the expected number of attacks for each heart and then add these together to get the final answer.

UwU

There are an endless number of ways Rengoku can reduce the health of a heart to 0 and after using a form, the health of the heart changes to a new smaller value (unless it is the first form). This motivates us to use dynamic programming. Let $DP[i]$ be the expected number of attacks to reduce a heart of health i to 0. Of course, $DP[0] = 0$. It follows that:

$$DP[i] = \sum_{F=1}^{K} (DP\left[\left\lfloor \frac{i}{F} \right\rfloor\right] + 1)/K$$

UwU

An issue arises though. What happens if $F = 1$? Let E be the expected number of attacks used after Rengoku finally chooses a form $F! = 1$.

Then the expected number of attacks is $E + (E + (K-1)/K)/K + (E + 2 \cdot (K-1)/K)/K^2 \ldots$ (He does not pick $F = 1$, he picks $F = 1$ one time with probability $1/K$, he picks $F = 1$ two times with probability $1/K^2$ and so forth). This is an arithmetico-geometric series and the sum ends up being:

$$(K \cdot E + 1)/(K-1)$$

You can use either a "double"geometric series or the formula for the sum of an infinite arithmetico-geometric series to find this final sum.

Alternatively (and more nicely), $DP[i] = E + (1 + DP[i])/K$, so we can find $DP[i]$ with simple algebra.

There are $O(N)$ states and $O(N)$ transitions, so we already have an $O(N^2)$ solution.

UwU

How can we do better? Well intuitively, $\lfloor \frac{N}{F} \rfloor$ seems like it takes on the same value quite a few times for large values of F. In fact, it only takes on $O(\sqrt{N})$ values! Let's try to prove this.

For the values $\ge \sqrt{N}$, we choose $F = 1, 2, \ldots, \lfloor \sqrt{N} \rfloor$ (if $N < \lfloor \sqrt{N} \rfloor \cdot (\lfloor \sqrt{N} \rfloor + 1)$, $\left\lfloor \frac{N}{\sqrt{N}} \right\rfloor < \sqrt{N}$, so handle this edge case). These are all the possible values $\ge \sqrt{N}$ since if we try $F = \sqrt{N} + 1$, $\lfloor \frac{N}{F} \rfloor < \sqrt{N}$.

Now for values $< \sqrt{N}$, realize we can cleverly choose $F = \left\lfloor \frac{N}{j} \right\rfloor$ for $j = 1, 2, \ldots, \lfloor \sqrt{N} \rfloor$ to attain values $1, 2, \ldots, \lfloor \sqrt{N} \rfloor$ (if N is a perfect square, discard the last value, $\lfloor \sqrt{N} \rfloor$ to avoid double count with the $\geq \sqrt{N}$ case). These are all the possible values $< \sqrt{N}$ since we attain every possible one. This is true because $\left\lfloor N / \left\lfloor \frac{N}{j} \right\rfloor \right\rfloor = j$ for $j < \sqrt{N}$. To show this, let $\left\lfloor \frac{N}{j} \right\rfloor = a$. Then N is some value from $a \cdot j$ to $a \cdot j + j - 1$. Thus, $\lfloor \frac{N}{a} \rfloor = j$ since $a \geq j$.

It turns out that $\left\lfloor \frac{N}{j} \right\rfloor$ is the largest F such that $\lfloor \frac{N}{F} \rfloor = j$. This is true because by definition of $\left\lfloor \frac{N}{j} \right\rfloor = F$, F is the largest number such that $F \cdot j \leq N$. This means the range from $F = \left\lfloor \frac{N}{j+1} \right\rfloor + 1 \ldots \left\lfloor \frac{N}{j} \right\rfloor$ will result in $\lfloor \frac{N}{F} \rfloor = j$.

Using these facts, we can optimize our dp to $O(N \cdot \sqrt{N})$!

UwU

Time complexity: $O(N \cdot \sqrt{N})$

Problem V. Thomas Game

All test data is randomly generated (despite the linked generator not compiling... my bad). The first thing you should strive to find out is how that helps. Well for most important thing here is that the convex hull of n randomly generated points on an infinite plane is rather small. I was under the impression it was around $n^{\frac{1}{3}}$ but it turns out $\log n$ is also pretty close. But with some stresstesting, the size of a convex hull of $5 \cdot 10^5$ points is around 40 at most.

Now comes the important observation. Lets define a ring (I am also informed that they are called onions...) as the convex hull of a set of points. Note that a ring is a set of points. Now let's define the ith ring as the ring of the array with the points from the first $i - 1$ rings removed. All points belong to a ring, and all rings are disjoint.

I claim that removing any 2 points from the array will always guarantee that the convex hull of whats left will only be made out of points from the first three rings.

By default the convex hull without any points removed only uses points from the first ring (well it is the first ring). To make the convex hull use points aside of the first ring, we have to remove one of the points from the first ring (I hope it make sense that removing points not on the first ring won't change the convex hull). For a similar reason, you have to remove the first point from the first ring and the second point from the second ring to give a chance of a point showing up on the third ring. So consider the following 4 cases for queries.

1. neither a or b is on the first ring. Output the convex hull area of the entire array

2. a and b are both on the first ring. The answer will only use points from the first and second rings

3. a is on the first ring and b is on the second ring. The answer will only use points from the first three points at worse

4. a is on the first ring and b is not on the first or second rings, the answer will only use points from the first and second rings, this is similar to case 2.

So you can just precompute the answers for

case 1: just find the ch of the entire array and store it

cases 2 and 3: for all pairs of points a and b such that a is the on the first ring and b is on either the first or second ring, find and store the ch without points a and b.

case 4: find the ch for all cases a is on the first ring, and you don't have to worry about b.

The total complexity is $O(n \log n + \log^3 n \cdot \log \log n + q \log n)$. $n \log n$ from finding the first three rings and the ch of the entire array, $\log^3 n \cdot \log \log n$ since there are $\log n \cdot \log n$ ways to pick one point from the first ring and one from the first/second rings and $\log n \cdot \log \log n$ time to find the ch, and $q \log n$ for the queries (binary search to find whether or not a point is in a ring), q is doable. Colin Galen also noted that with

the three rings idea, you can just find the ch each time for a total of q times, and since you dont have to sort the points repeatedly, $O(q \log n)$ is doable.

The implementation is very kactl-heavy. I would not have set this problem if copying code was not allowed :). you can find my implementation and a better version of the editorial in github /thomasgame

Problem W. Sol's Problem Meeting

Subtask: Find the minimal maximum angriness given that the path length must be at most K We simply do binary search on the angriness, and use dijstkra to determine whether it is possible to get a path length less than or equal to K. We do $N \log N$ dijsktra every iteration of binary search, so the complexity for this subtask is $N \log^2(N)$

Solution for the Problem: The tricky part is that one edge can be deleted and we must look for the worst possible scenario. Assume for now that no edge is deleted; call the main path from node 1 to node N m. To get our desired answer, we wish to go through each edge on m and compute the minimum path length that doesn't use that edge. We then take the maximum out of all of these lengths, and check if it is at most K.

Look at another edge e not in m; note that the shortest path from 1 to N through e will use a prefix and suffix of m. In other words there is an interval of edges in main path which the alternate path through e does not use. The reason for this is because it is never optimal to diverge from m and then come back unless it is to include e. Call this interval which the path through e doesn't go through m the "interval of divergence"or IOD, which can be found for each edge using a second dijsktras.

We must iterate through all the edges not on m, and get the IOD for each edge along with the minimum path length to go through that edge from 1 to N. To get the minimum length of a path not using an edge f in path m, we must query for each edge e not in m which has an IOD containing f (i.e. the path through e doesn't use edge f). Iterating through each edge and checking its IOD is obviously too slow, so we use a segment tree or a set to store all this information. Each update will be a range and specify a minimum path length over that IOD. We will then have point queries, where we look at each interval and see if it includes the queried edge, and get the minimum of them. We do this with every edge on m.

Updating and checking the IOD is $\log N$ per edge and there are at most M operations per binary search, the complexity is $N \log^2(N) + M \log^2(N)$.

Problem X. Endeavor's Agency

The problem asks us to find the size of the minimum vertex cover for the graph. However, minimum vertex cover is NP-hard, so there must be something special about this graph. Perhaps we can convert the problem from a graph problem to something else.

UwU

In order to do this, the graph must represent something special, so let's continue to transform it and see if we can end up with anything. We know that if the sequence of cities on the path from u to v is increasing, then there is an edge $u-v$. Let's make graph directed so that the road between u and v is now a directed road from u to v, where $u < v$. Then this property is still satisfied, because the edges point to larger values, and the property of the graph is about increasing cities on paths. Additionally, lets assign a "$time$"to each node such that $time(u) > time(v)$ where $u < v$. The property is still satisfied since the edges point back in time and paths also go back in time as they are traversed. Realize now that if instead of a "time"it was an index, the edges in the graph would be inversions of some permutation. In formal terms, the graph is both a comparability graph and a permutation graph.

UwU

Can we retrieve the permutation from the graph? Well, the edges give us an idea about the relative positions of different numbers. An edge $u \to v$ means that u must come after v. There are alot of missing edges though. Based on the fact that edges currently represent inversions, if there is not an edge between u and v, where $u < v$, then u comes before v. Thus, lets also add in an edge from v to u, where $u < v$, if

there is not an edge between the two. Let new edges be the set of these edges. Now the topological sort of the graph will give us a unique permutation because edges point to numbers that come before the current number. So, when a number is pushed onto the stack that represents the permutation, all numbers that were supposed to come before it were already pushed onto the stack. Of course, there are $O(N^2)$ edges, so we need a faster algorithm. There are two options.

The first is that we can sort the numbers $1 \dots N$ since we can find out if u comes before v in $O(1)$ time for the comparator. This takes $O(N \cdot log(N))$ time.

The second is that number u is at the $outdegree(u) + 1$th position since $outdegree(u)$ numbers come before it. $Outdegree(u)$ is the number of $v > u$ that come before u plus the number of $v < u$ that come before u. Thus, $outdegree(u)$ is also the outdegree for u before we added new edges plus $u - 1-$ indegree for u before we added new edges. This takes $O(N + M)$ time.

UwU

Now that we know the graph represents a permutation, lets see if this transformation helps to solve the problem. A vertex cover of the graph means there does not exist an edge $u - v$ such that both u and v are not in the set. What does this mean for the permutation? Well it means that among the numbers not in the set, there cannot be any inversions. Thus, a vertex cover is any set of nodes such that the remaining set of nodes not in the vertex cover form an increasing subsequence! Therefore, N = size of vertex cover + length of remaining increasing subsequence. Which is the same as size of vertex cover = $N-$ length of remaining increasing subsequence. Thus, if we find the longest increasing subsequence, we can find the minimum vertex cover! This takes $O(N \cdot log(N))$ using a standard algorithm.

UwU

Time complexity: $O(N \cdot log(N) + M)$

Problem Y. Lattice MST

disclaimer: i will have a lot of off by ones in this editorial... refer to implementation for those :)

So I assume you have already read the statement.. if not, go do that now. So now try to solve it for $d = k = 2$, and $D[1] * D[2] \leq 10^6$. (i want a linear sol). The important observation here is that the furthest point from any given point is one of the corners. so if we have our corners then we can loop over each point and draw an edge to the furthest point from it (which is one of 4 points) in $O(D[1] * D[2])$ time. Note that there is actually an edge case since the edges between corners are duplicated, but we will handle that later. So lets define S as the sum of the distances between all nodes and their furthest corners.

so,

$$S = \sum_{i=1}^{n} \sum_{j=1}^{m} \max(i, n-i)^2 + \max(j, m-j)^2$$

Now we can evaluate S in $O(D[1] * D[2])$ time. Note that i didn't actually loop over the corners, but instead just took the largest direction it can go and squared it. Now the observation is, "oh wait i can seperate the sigmas"and, yes, you can. so we can rewrite S

$$S = m \cdot \sum_{i=1}^{n} max(i, n-i)^2 + n \cdot \sum_{j=1}^{m} max(j, n-j)^2$$

This is now evaluating S is $O(D[1] + D[2])$ time. now comes the second observation, which is "oh wait, i dont need the sigmas... i can use sum of squares to find this sum and, yes, you are right. with some casework on when n is odd and even, the sigma expands to something like $n^2 + (n-1)^2 + (n-2)^2 \cdots (n/2+1)^2 + (n/2)^2 + (n/2+1)^2 \cdots (n-2)^2 + (n-1)^2 + (n)^2$ or it goes down to $(n/2)^2$ and back up again. so if we have $sumsq(n) = \frac{(n)(n+1)(2n+1)}{6}$ or our favorite sum of squares formula, the answer looks something like

$$S = m \cdot (sumsq(n) - sumsq(n/2)) \cdot 2 + n \cdot (sumsq(m) - sumsq(m/2)) \cdot 2$$

Now we have S evaluated in $O(1)$ time. I (chessbot), initially proposed this problem with $d = k = 2, D[1] * D[2] \leq 10^6$ and PurpleCrayon made the above observations in around 10 minutes, and you can blame him for having to generalize this problem to d dimenions and k powers for distance :).

Now comes the first of two fun parts, generalizing the distance formula. So now lets keep $d = 2$ but k can now be ≤ 500 and $D[1], D[2] \leq 10^9$. Now we can define $kthsum$.

$$kthsum(n, k) = \sum_{i=1}^{n} i^k$$

So we can see that $kthsum(n, k)$ and be found in $O(n \log k)$ time by simply looping over n and using binary exponentiation. Note that $O(n)$ time with $O(n)$ memory is also doable with seive :). But in this context, n is $D[i]$ which is 10^9 and then we tle. So the idea is to make a $k+1$ -degree polynomial that uniquely defines the set of points $\{(1, 1^k), (2, 1^k + 2^k), (3, 1^k + 2^k + 3^k), \cdots, (k+2, 1^k + 2^k + \cdots (k+2)^k)\}$. We know that the sum of k powers can be expressed as a polynomial that grows relateive to $n^{(k+1)}$ (proof is left as an exercise to the reader), and we also know that an n-degree polynomial is unique defined by $n+1$ points. Now we can find a $k+1$-degree polynomial that goes through the first $k+2$ points of $i, kthsum(i, k)$. This can be done in $O(k \log mod)$ or $O(k)$ depending on the implementation. https://codeforces.com/blog/entry/82953 and https://codeforces.com/blog/entry/23442 are good resources on this.

Now lets define a new function, $eval(n, k)$. which is basically $\sum_{i=1}^{n} max(i, n-i)^k$ but evaluated faster since it looks more like

$$eval(n, k) = 2 * (kthsum(n, k) - kthsum(n/2, k))$$

with some edge cases concerning even/odd n, but $eval(n, k)$ can be evaluted in $O(k \log mod)$ time. Now lets try to generalize or 2d approach into 3d. Consider the contribution of $eval(D[1], k))$ for some $D[1]$ and some k. If $d = 2$ then the contribution is just $D[2]$ since there are $D[2]$ "rows"that use the sum of $eval(D[1], k)$. Now lets say $d = 3$, the new contribution is just $D[2] * D[3]$ since there are that many "rows"in the 3d figure. Try to draw it out on paper, it will help. Now with this, let's redefine S.

$$S = \sum_{i=1}^{d} (\prod_{j=1, j \neq i}^{d} D[j]) \cdot eval(D[i], k)$$

or the product of all the dimensions save $D[i]$ multiplied by $eval(D[i], k)$. The important observation here is that S can be evaluated in $O(d \cdot k \log mod)$ time. which is enough to fit within the limits. So are we done? no, we stil have not handled the case for corners sharing edges.

Now we have S but it turns out S is wrong as the answer to our MST. So for the most basic case, we will solve the case of corners for $d = 2$. Let's define $n = D[1], m = D[2]$. So we have two edges from $(1, 1) \to (n, m)$ and two from $(1, m) \to (n, 1)$ since the corners are the furthest nodes from each other. Now we should subtract $2 * (n-1)^2 * (m-1)^2$ or the weights of two of the corner edges and add one more edge in that is legal. Now i suggest you grab a sheet of paper since it will help. Imagine a 4×4 lattice. the edges will look something like edges from each corner to the nodes near the opposite corner. So edges out of $(1, 1)$ will end at nodes close to $(4, 4)$. But how close? around $4/2$ close or just 2 nodes close. Since any further and it would be just as (if not more) optimal to draw your edge out from another corner. More specifically, the edges from $(1, 1)$ go to the nodes between $(n/2, m/2)$ and (n, m) with some off by ones concerning rounding $n/2$ and $m/2$. So the furthest node from $(1, 1)$ that isn't directly counted is either node $(n/2, m)$ or node $(m/2, n)$ (proof is left as an exercise to the reader). So in the end if we have S already, we should subtract $(n-1)^2 + (m-1)^2$ twice and add $\max((n/2)^2 + m^2, (m/2)^2 + n^2)$. note there are off by ones. Now lets try this on 3d, imagine a $3 \times 3 \times 2$ grid. It gets a little scuffed, but on a 3 dimensional grid, the furthest point not already connect is $\max((D[i]/2)^k + \sum_{j=1, j \neq i}^{d} D[j]^k)$ across all i. Proof is left as an exercise to the reader. Now try to see by always connecting the furthest legal edge doesn't work. So on the $3 \times 3 \times 2$ board, we have that the best edge is between corners on a 3×3 face. But cant draw more than 2 of them before a cycle appears. So you have to take 1 of the second best legal edge. So the result is a formula that works for d dimensions. Let's define C as the sum of all of

the "legal"edges we are taking after removing the duplicate corners pair edges. If D is sorted in least to greatest order

$$C = \sum_{i=1}^{d-1} 2^{i-1} \cdot ((D[i]/2)^k + \sum_{j=1, j\neq i}^{d} D[j]^k)$$

which is taking as many of the largest legal edge as possible, then as many of the second largest. This formula can be proven with something like kruskal's. So the our end answer is

$$S - C - (2^{d-1} * (\sum_{i=1}^{d} D[i]^k))$$

So we have S and we subtract C and the last term is the amount of duplicate diagonal or edges between pairs of corners multipled by the weight of such a corner edge. This entire thing is evaluated in $O(d \cdot k \cdot mod)$ time or $O(d \cdot k)$ depending on the implementation. Thank chessbot for the good problem and blame purple for the bad problem. The code can be found on the github under /latticemst/model.cpp. also the editorial in the git is much nicer than this one.

Good luck upsolving!