Problem A. Automatic Sprayer 2

Problem idea and preparation: Suchan Park (tncks0121)
First solver: Past Glory : Pavel Kunyavsky, Artem Vasiliev, Gennady Korotkevich (0:41)
Total solved team: 47

$$
\begin{aligned}
E_{i,j} &= \sum_{x=1}^{n}\sum_{y=1}^{n} A_{x,y}\cdot(|i-x|+|y-j|)\\
&= \sum_{x=1}^{n}\left\{\left(\sum_{y=1}^{n} A_{x,y}\right)\cdot|i-x|\right\}+\sum_{y=1}^{n}\left\{\left(\sum_{x=1}^{n} A_{x,y}\right)\cdot|y-j|\right\}
\end{aligned}
$$

Define $R_x := \sum_{y=1}^{n} A_{x,y}$, $C_y := \sum_{x=1}^{n} A_{x,y}$ to simplify:

$$E_{i,j} = \sum_{x=1}^{n}(R_x\cdot|i-x|)+\sum_{y=1}^{n}(C_y\cdot|y-j|)$$

We can see that E only depends on $R_{1..n}$ and $C_{1..n}$, so let's determine them first.

Let's introduce a function $f_k(t)=|t-k|$. We know that $E_{i,j}=\sum_{x=1}^{n} f_x(i)R_x+\sum_{y=1}^{n} f_y(j)C_y$ holds.

The slope of the graph of the function $y=f_k(t)$ changes only at $t=k$ (-1 at $t\to k-$ and $+1$ at $t\to k+$), so we can kind of say $f''_k(t)=\begin{cases}2 & \text{if } x=k\\ 0 & \text{if } x\neq k\end{cases}$ holds.

If t and k are both integers, $\lim_{u\to t-} f'_k(u)=f_k(t)-f_k(t-1)$ and $\lim_{u\to t+} f'_k(u)=f_k(t+1)-f_k(t)$ both holds, so we can define $f''_k(t) := (f_k(t+1)-f_k(t))-(f_k(t)-f_k(t-1))$.

Since $f''_k(t)$ is an indicator function of k over $\{1,2,\cdots,n\}$, we focus on adding and subtracting $E_{i,j}$s properly to see the indicator function inside the summation:

$$
\begin{aligned}
&(E_{k+1,1}-E_{k,1})-(E_{k,1}-E_{k-1,1})\\
=&\sum_{x=1}^{n}\{(f_x(k+1)-f_x(k))-(f_x(k)-f_x(k-1))\}\cdot R_x\\
=&\sum_{x=1}^{n} f''_x(k)\cdot R_x = 2\cdot R_k
\end{aligned}
$$

Thus, for each $2\le x,y\le n-1$, it turns out

$$R_x=((E_{x+1,1}-E_{x,1})-(E_{x,1}-E_{x-1,1}))/2$$

$$C_y=((E_{1,y+1}-E_{1,y})-(E_{1,y}-E_{1,y-1}))/2$$

holds.

Knowing the exact values of $R_{2..n}$ and $C_{2..n}$, we can find out these equations about R_1, C_1, R_n and C_n.

$$R_1 + C_1 = \left(E_{n,n} - \sum_{x=2}^{n-1} (|x-n| \cdot R_x) - \sum_{y=2}^{n-1} (|y-n| \cdot C_y)\right) / (n-1)$$
$$R_1 + C_n = \left(E_{n,1} - \sum_{x=2}^{n-1} (|x-n| \cdot R_x) - \sum_{y=2}^{n-1} (|y-1| \cdot C_y)\right) / (n-1)$$
$$R_n + C_1 = \left(E_{1,n} - \sum_{x=2}^{n-1} (|x-1| \cdot R_x) - \sum_{y=2}^{n-1} (|y-n| \cdot C_y)\right) / (n-1)$$

$$R_1 + R_n + \sum_{x=2}^{n-1} R_x = C_1 + C_n + \sum_{y=2}^{n-1} C_y$$

These equations are enough to uniquely determine R_1, C_1, R_n and C_n.

After knowing $R_{1..n}$ and $C_{1..n}$, there are several possible ways to find any possible matrix A with predetermined row sum and column sum. The simplest way is the following:

```
for (x in 1..n) for (y in 1..n) {
    A[x][y] = min(R[x], C[y]);
    R[x] -= A[x][y]; C[y] -= A[x][y];
}
```

The proof is that each time `A[x][y]` is positive, at least one of R_x or C_y becomes zero, and we can apply mathematical induction on "the number of non-zero R_x and C_ys". Or we can think of this procedure as running Ford-Fulkerson on obvious flow graph.

Shortest solution: 796 bytes

Problem B. Cilantro

Problem idea and preparation: Dongjae Lim (doju)
First solver: japan02 : Shigemura, Kawasaki, Yui Hosaka (1:02)
Total solved team: 7

Let's solve an equivalent decision problem, where you determine if you can put at most x dishes in the stack before any customers arrive. If you can put at most x dishes into the stack, then you have a strategy that can serve $1, 2, \ldots, x$-th noodles to the first customer as long as the types are matched.

To solve this decision problem, a simple greedy strategy suffices. Start by putting the first x noodles in the stack. For each customer, cook more noodles until the top of the stack matches the preference, and serve the noodles if it matches. This greedy strategy can be implemented in $O(n)$ time, which gives a $O(n \log n)$ algorithm. As intended, no teams succeeded to push this into the time limit.

We can prove the correctness of the greedy strategy. Moreover, the greedy strategy can always find a solution.

Theorem 1. *Given that $x = 0$, the greedy strategy always finds a correct serving scheme, if $|S| = |T| = n$ and the number of* `Y` *and* `N` *in S and T are equal.*

Proof. We prove this by induction on n. The case with $n = 1$ is trivial. Suppose that there exists some $1 \leq i \leq n - 1$ such that $S[1, i], T[1, i]$ have a same occurrence count of `Y` and `N`. Then, by inductive hypothesis, the strategy for $S[1, i], T[1, i]$ and $S[i + 1, n], T[i + 1, n]$ exists. Otherwise, $S[1] = T[N]$, and consequently $S[2, n], T[1, n - 1]$ have a same occurrence count, which means you can put the first noodle, apply induction hypothesis for $n - 1$ and serve it to the final customer. □

It is important to notice that there is complete freedom over which i to match, as long as the occurrence count is preserved.

Putting the first x noodles in a stack implies that the first $x - 1$ noodles can avoid being served to the first customer. For each noodle in order, we decide the matching customer so that we serve the first customer as late as possible. To do this, it's not worse to pick the largest i that is matchable, and if we are only left with the option of $i = 1$, then we can declare the answer. As a result, we obtain a different greedy algorithm that determines the maximum possible x in linear time.

Shortest solution: 401 bytes

Problem C. Equivalent Pipelines

Problem idea and preparation: Changki Yun (TAMREF)
First solver: Zagreb Oblutci (0:21)
Total solved team: 64

Remark that $v_T(i,j)$ is the minimum edge weight on the only path connecting i and j. We want to determine the correspondence of $N(N-1)/2$ tuples $(i,j,v_T(i,j))$. To overcome the time limit, one can think of hashing as a hack.

Pick two random functions f, g and a random prime P. Then compute the hash value

$$h_T = \sum_{i<j} f(i)f(j)g(v_T(i,j)) \mod P$$

and group the trees by hash value. If P is sufficiently large (empirically, $\sim 10^{18}$), it is enough to avoid hash collision.

Using union-find technique, one can easily compute h_T in $O(n \log n)$ time, resulting in time complexity $O(dn \log n)$.

There is also a deterministic solution. We think of the "merge sequence" in the union-find procedure. Sort the edge in the non-increasing order of weight and simulate the union-find operation. If an edge with weight w connects components A and B, $v_T(a,b) = w$ for all $a \in A$ and $b \in B$. Thus, for all weights w, the trees in the same group must share the same components "merged by weight w".

Suppose a component a is merged into a larger component a' by an edge weight w. Then using a tuple (w, a, a') for all merge events is sufficient to represent a tree. There are only $O(n)$ important tuples, so comparing such tuples by a trie or std::map is enough to get accepted in $O(dn \log n)$ time.

Shortest solution: 1745 bytes

Problem D. Flowerbed Redecoration

Problem idea and preparation: Joonpyo Hong (spectaclehong)
First solver: japan22 : goodbaton, Tatsuhito Yamagata, Yasunori Kinoshita (2:36)
Total solved team: 4

Operations that rotate the region $d \times d$ can be expressed as permutations, and permutations can be thought of as one-to-one correspondences.

Let F be a function that rotates the fixed top-left area, and a G be a function that brings the next target area to the top-left: This can be easily defined as a shifting permutation. A function $H = G \circ F$ rotates the area and brings the next target area to top-left. If we can do this quickly $k = (M - d)/x$ times, we can process the first row. Then we shift the grid downwards by y and repeat the procedure to solve the problem.

This can be solved in $O(n \log k)$ by implementing a divide-and-conquer-based fast exponentiation, or in $O(n)$ by decomposing the permutation into cycles and rotating each cycle k times.

Shortest solution: 1054 bytes

Problem E. Goose Coins

Problem idea and preparation: Dongkyu Han (queued_q)
First solver: USA1 : Kevin Sun, Scott Wu, Andrew He (0:27)
Total solved team: 44

First of all, handle the impossible case where p is not a multiple of c_1.

Consider the following greedy algorithm to make p goose-dollars. Select the n-th coin as many as possible so that the total value does not exceed p. Then select the $(n-1)$-th coin as many as possible so that the total value does not exceed p. Repeat this until the total value reaches p. We will call the resulting set of coins the "base solution." Let's say that there is a_i number of i-th coins in the base solution.

Now consider another set of coins that sums to p goose-dollars. If this set of coins is different from the base solution, we can show that there exists $i(< n)$-th coin that is used c_{i+1}/c_i or more times. In other words, if every i-th coin is used less than c_{i+1}/c_i times, this set of coins is the same as the base solution.

The proof is as follows. If every i-th coin is used less than c_{i+1}/c_i times, the sum of the values of $1, 2, \cdots, (n-1)$-th coins in the set is less than c_n. So we have to use the n-th coin as many as possible to make p goose-dollars, since otherwise we cannot cover the rest of the price with $1, 2, \cdots, (n-1)$-th coins. By the same logic, we have to use the $(n-1)$-th coin as many as possible to make the remaining price, and the same is true for $(n-2), \cdots, 1$-th coins. So it becomes equivalent to the base solution.

Using this fact, you can replace c_{i+1}/c_i coins of some i-th type with a single $(i+1)$-th coin in any solution other than the base solution. This process reduces the total number of coins, so we eventually reach the base solution by repeating the process. In other words, we can make any solution from the base solution by repeatedly replacing some i-th coin into c_i/c_{i-1} coins of $(i-1)$-th type. We'll refer to this process as "coin splitting."

From here, there are two ways to solve the problem.

Solution 1. (by queued_q) We have to make k coins in total by splitting the coins in the base solution. Suppose that, for every coin in the base solution, we know the minimum weights of $1, \cdots, k$ coins that were generated from splitting the coin. We can combine them to get the minimum weights of $1, \cdots, k$ coins that sum to p goose-dollars.

Formally, let $W_x[1..k]$ be an array that contains the minimum weights of $1, \cdots, k$ coins that sum to x goose-dollars. Then we can calculate W_p by combining W_{c_i}'s. Assume that the number of coins in the base solution is m in total, and each of their values is d_i. Then, it holds that

$$W_p[j] = \min_{l_1+\cdots+l_m=j} W_{d_1}[l_1] + \cdots + W_{d_m}[l_m].$$

The above operation, which combines several W_x's, can be further simplified into the process of repeatedly combining two W_x's. Define the binary operation $\oplus$ on the minimum-weights arrays A and B as follows:

$$(A \oplus B)[j] := \min_{a+b=j} A[a] + B[b].$$

Intuitively, it represents a situation where we use a coins from the A array and b coins from the B array to make j coins in total. This operation takes $O(k^2)$ time. Also, note that $\oplus$ is associative and commutative.

Now we can represent W_p as follows.

$$W_p = \underbrace{(W_{c_1} \oplus \cdots \oplus W_{c_1})}_{a_1 \text{ times}} \oplus \cdots \oplus \underbrace{(W_{c_n} \oplus \cdots \oplus W_{c_n})}_{a_n \text{ times}} = (W_{c_1})^{a_1} \oplus \cdots \oplus (W_{c_n})^{a_n}.$$

If we know every W_{c_i}, we can calculate $(W_{c_i})^{a_i}$ quickly by applying exponentiation by squaring. The number of operation is $\sum_i \lg a_i \le \sum_i [\lg(c_{i+1}/c_i)+1] = \sum_i (\lg c_{i+1} - \lg c_i) + n \le \lg p + n = O(\lg p)$, so we can calculate W_p in $O(k^2 \lg p)$ time.

Now we have to figure out how to calculate W_{c_i}. To make c_i goose-dollars, we can use one i-th coin or split it. If we split it once, we get c_i/c_{i-1} coins of $(i-1)$-th type. Since we can split them further, it holds that

$$W_{c_i}[j] = \begin{cases} w_i & \text{if } j = 1 \\ (W_{c_{i-1}})^{c_i/c_{i-1}}[j] & \text{if } j > 1 \end{cases}.$$

It takes $O(k^2 \lg p)$ time to get all W_{c_i}'s by a similar logic. Hence the total time complexity to calculate W_p is $O(k^2 \lg p)$. The minimum total weight of k coins is $W_p[k]$. If it is infinity, there is no solution. We can also calculate the maximum by flipping the signs of the weights and applying the same algorithm.

Solution 2. (by tncks0121) Consider the process of splitting coins from n-th to first. Define the DP table as follows.

$$D[i,j,l] := (\text{The minimum weight of a set of coins that sum to } p\text{, where some of } (i+1), \cdots, n\text{-th coins were split, there are } j \text{ coins in total, and there are } l \text{ coins of } i\text{-th type.})$$

If we think about the process of splitting m coins of i-th type in the set of coins that $D[i,j,l]$ represents, we can obtain the following relation. Here $b_i = c_i/c_{i-1}$ represents the number of $(i-1)$-th coins that gets created when we split an i-th coin.

$$D[i-1, j+(b_i-1)m, a_{i-1}+b_i m] = \min_{m \le l \le k} \{D[i,j,l]\} + (b_i w_{i-1} - w_i)m.$$

We only have to consider the cases where the total number of coins does not exceed k. So we ignore the cases where $j+(b_i-1)m$ or $a_{i-1}+b_i m$ exceed k. We can calculate the suffix minimums to get $\min_{m \le l \le k}\{D[i,j,l]\}$ in $O(1)$, so each $D[i,j,l]$ can be filled in $O(1)$. The total time complexity is $O(nk^2)$, which is same as the number of the DP states. The minimum total weight of k coins is $\min_l D[1,k,l]$. If it is infinity, there is no solution. We can also calculate the maximum by flipping the signs of the weights and applying the same algorithm.

Shortest solution: 1776 bytes

Problem F. Hedgehog Graph

Problem idea and preparation: Yeonghyun Kim (kipa00)
First solver: japan17 yosupo (4:41)
Total solved team: 1

1. Solution with $2\sqrt{V} + \log V$ queries. We will only make queries that starts at vertex $s = query(1, 10^6)$, since s lies in the cycle. For all $1 \le i \le 1\,000$, ask $query(s, i)$ and $query(s, 1000i + 1)$. Among the 2000 queries, there exists a vertex that was returned twice. Let x, y be such query with $query(s, x) = query(s, y)$. The answer is a smallest divisor d of $|y - x|$ with $query(s, d) = s$.

If $query(s, d) = s$, then for all integer $k > 1$, $query(s, kd) = s$. As a result, you don't have to try all divisors. Take a prime factorization of $|y-x|$, and try dividing to each factor in the multiset, taking the divided result if it is still a multiple of the period.

Abstracting the first solution. Taking all the assumption at the first solution, we can reformulate the problem. We need to find a short sequence $x_1, x_2, \ldots, x_n$, such that for any number $1 \le k \le 10^6$, there exists a pair x_i, x_j where $|x_i - x_j|$ is a non-zero multiple of k. For example, the solution 1 generates the length-2 000 sequence

$$x_i = \begin{cases} i & \text{if } i \le 1\,000 \\ 1000(i - 1000) + 1 & \text{if } i > 1\,000 \end{cases}.$$

And for every number $1 \le k \le 10^6$, $|x_{\lceil k/1000 \rceil + 1000} - x_{(-k) \bmod 1000 + 1}|$ is a nonzero multiple of k. The total query spent by this approach is $1 + n + \log \max(x_i)$.

Given that the sequence x_i is generated, the only algorithmically hard part is factorization. Here, you can use standard algorithms like Pollard-Rho, or even try a trivial algorithm that tries all factors $\le 10^6$ since any larger prime factors don't contribute to the answer.

2. Solution with $\sqrt{V} + 2\log V$ queries.

One idea is to observe that we only need to consider the number $500\,001 \le k \le 10^6$. Since all numbers $1 \le k \le 500\,000$ have a multiple in the range $[500\,001, 10^6]$, finding their multiples are enough.

The first construction generated two sequence $A = [1, 2, \ldots, 1000], B = [1001, 2001, \ldots, 10^6 + 1]$ where $|A_i - B_j|$ covers all multiple of k, and simply concatenated it. We will also use such strategy, but we will cover two elements at once.

Let $U = 500\,000$ and $L = 1\,000$. we will construct a sequence A, B such that $|A_i - B_j|$ is in a form of $M =$

$$\begin{bmatrix} (U+1)(U+L) & (U+2)(U+L-1) & \ldots & (U+L/2)(U+L/2+1) \\ (U+L+1)(U+2L) & (U+L+2)(U+2L-1) & \ldots & (U+3L/2)(U+3L/2+1) \\ \vdots & \vdots & \ddots & \vdots \\ (U+L^2/2-L+1)(U+L^2/2) & (U+L^2/2-L+2)(U+L^2/2-1) & \ldots & (U+L^2/2-L/2)(U+L^2/2-L/2+1) \end{bmatrix}$$

In this 500×500 matrix, $M_{i,j+1} - M_{i,j} = L - 2j$, $M_{i+1,j} - M_{i,j} = 2LU + 2iL^2$. Thus, we can find a sequence a, b such that $a_i - b_j = M_{i,j}$ Using this fact, you can find a length-1 000 sequence where $max(x) = O(V^2)$.

3. Solution with $\sqrt{2/3V} + 3 \log V$ queries.

We can slightly improve the previous idea. The matrix above illustrates a way to construct a sequence a_i, b_j such that $a_i - b_j$ covers two elements of the arithmetic progression. Thus we can find a sequence a, b of length $\lceil \sqrt{166\,667} \rceil = 409$ to cover an arithmetic progression of $333\,335, 333\,337 \ldots 999\,999$.

This covers all odd numbers, since all numbers $1 \leq 2k+1 \leq 333\,333$ have a odd multiple over that arithmetic sequence. Every even number at most 10^6 can be represented as $n = 2^k \times (2a+1)$ where $2a+1 \leq 10^6, 1 \leq k \leq 19$. By multiplying all matrix entities by 2^{19}, we can cover all numbers with $max(x) = O(V^3)$.

Challenge. Given that we have no restriction on $max(x)$, can we devise a generalized construction that minimizes the queries, asymptotically or exact?

Shortest solution: 1145 bytes

Problem G. Lamb's Respite

Problem Idea: Jaehyun Koo (koosaga)
Preparation: Jaeung Lee (L0TUS)
First solver: USA1 : Kevin Sun, Scott Wu, Andrew He (1:04)
Total solved team: 18

Without the ultimate ability, let's consider the condition where the player may die. Let p be the action where the player achieved maximum HP afterward, and q be an action where the player died afterward. For the player to die, $a_{p+1} + a_{p+2} + \ldots + a_q \le -H$ should hold. In other words, if the player dies, there exists a subarray where its sum is at most $-H$.

Conversely, suppose that the minimum sum subarray of a_i is $a_{l+1}, \ldots, a_r$, and its sum is at most $-H$. Since the subarray has the minimum sum, there exists no position where $a_{l+1} + \ldots + a_j > 0$ for $l+1 \le j \le r$. Thus, there exists no cases where the champion *overheals*. The champion will get at least H damages without exception, and will die regardless of its starting health.

What happens when the *Lamb's Respite* ability is on, is very similar to the usual cases. If the player reached the HP $\lceil \frac{H}{10} \rceil$, then the player stays in that health until its deactivation. So it can be considered as death, although it doesn't kill the champion.

We can easily maintain the minimum prefix, suffix, subsegment sum in a segment tree. As a result, the above observation gives an algorithm to determine whether the champion dies or not in the interval $[1, i-1]$. This extend to interval $[i, j]$ and $[j+1, n]$ as well. However, we don't start with full HP in those cases. We need to know the resulting HP after processing each interval.

Using the same argument as above, you can show that the resulting health from interval $[1, i-1]$ equals to $H' = H+$ (minimum suffix sum of the interval $[1, i-1]$). The initial HP can be supplied to the interval $[i, j]$ as well by appending a single integer $H' - H$ at the front. (The usual way of implementing the segment tree will make this part of implementation very straightforward).

The case $[j+1, n]$ is solved similarly. Thus we can determine the resulting HP. The time complexity of this solution is $O((n+q)\log n)$.

Shortest solution: 1726 bytes

Problem H. Or Machine

Problem idea and preparation: Jaemin Choi (jh05013)
First solver: Past Glory : Pavel Kunyavsky, Artem Vasiliev, Gennady Korotkevich (0:10)
Total solved team: 70

When dealing with bitwise operations, it's often useful to treat a b-bit integer as a tuple of b independent 1-bit integers (0 or 1). Using this approach, we will assume each register value is 1-bit, and solve the problem 8 times to get the final answer.

Notice that once a register value becomes 1, it stays at 1 forever. For each register i, let T_i be the number of operations required to make the register i's value equal to 1 (or $T_i = 0$ if it's already 1 before starting any operation). Then the value after t operations is 1 if $T_i \leq t$, and otherwise 0. Now the problem reduces to computing T_i for each i.

This can be modeled as a graph problem. Treat each register as a vertex, and each operation as an edge. The i-th operation represented by a and b is modeled as an edge from b to a with "index" i. If b becomes 1 at time T_b, then it propagates its value to a at time T', where T' is the smallest number $> T_b$ such that $T_a \equiv i - 1 \pmod{l}$.

Now this looks like a shortest path problem in which Dijkstra's algorithm can be used. However, the distance metric is unusual: instead of adding the weight of an edge, we are computing some value T' that depends on the edge and the distance to the previous vertex, T_b. Nevertheless, Dijkstra's algorithm still works correctly. To see why, note that the proof of correctness of Dijkstra's algorithm depends only on the fact that the distance does not decrease by taking an alternative path; the exact distance formula does not matter as long as $T' \leq T_b$.

The total time complexity is $O(bl \log n)$, where b is the number of bits (which is 8).

As a final note, this approach can be further generalized: define a distance metric $T(t, e)$, which means that if we take an edge e at time t, we arrive at the other vertex at time $T(t, e)$. If $T(t, e) \geq t$ is guaranteed, Dijkstra's algorithm can be applied. In the original Dijkstra's algorithm, we set $T(t, e) := t + e_w$ where e_w is the weight of e. For another example, $T(t, e) := \max(t, e_w)$ gives a minimum bottleneck path: a path whose maximum edge weight is minimized.

Shortest solution: 925 bytes

Problem I. Organizing Colored Sheets

Problem idea and preparation: Hyunuk Nam (jwvg0425)
First solver: japan02 : Shigemura, Kawasaki, Yui Hosaka (4:32)
Total solved team: 2

Let's call the tuple (W, H, i, j) *inadmissible* if the colored sheet of size $W \times H$ can not cover the uncolored cell (i, j). For convenience, denote the tuple (W, H) *inadmissible* if there exists a cell (i, j) where (W, H, i, j) is inadmissible. You can first observe that if (W, H) is inadmissible, $(W, H + 1)$ and $(W + 1, H)$ is inadmissible.

We introduce an important fact for inadmissible tuples.

Theorem 2. *If (W, H) is inadmissible, there exists some cell (i, j) such that (W, H, i, j) is inadmissible and (i, j) is either adjacent to a colored cell or outside of the grid.*

Proof. Suppose not. For a size (W, H), denote the cell (i, j) *bad* if (W, H, i, j) is inadmissible. Consider some 4-component of bad cells C. As the theorem is false, all cells adjacent to C are good cells. Take one such cell (p_1, q_1) and consider a $W \times H$ rectangle R that covers (p_1, q_1). By definition, there exists a cell $(p_1 + dx, q_1 + dy) \in C$ where $(dx, dy) \in \{(0, 1), (1, 0), (0, -1), (-1, 0)\}$.

Now let R' be a rectangle where R is pushed toward (dx, dy). Then, $R' - R$ is either a length W row or length H column. Suppose that there is a colored cell $(p_2, q_2) \in R' - R$. $(p_1 + dx, q_1 + dy)$ is in $C \cap (R' - R)$. Since any cell in C can't be adjacent to a colored cell, there exists some uncolored good cell between $(p_1 + dx, q_1 + dy)$ and (p_2, q_2). If you take the rectangle covering that good cell, it consequently covers $(p_1 + dx, q_1 + dy)$ due to the existance of colored cells in the right. We reach a contradiction. □

The theorem enables us to view the problem in another direction. Instead of enumerating all (W, H) to find bad (i, j), you can enumerate all colored cells and directions to find inadmissible (W, H) adjacent to such cells. Let's fix the direction to $(-1, 0)$ and solve the problem since the other case can be dealt with rotations.

One naive solution is to fix the cell (i, j) where $(i + 1, j)$ is either colored or outside the grid, and find all admissible sizes instead. Maintain the value $H_{i,j}$ =(the number of consecutive uncolored cells in upper direction), and enumerate all admissible sizes by starting from $H_{i,j} \times 1$, gradually reducing the heights while increasing the widths with two pointers. You can enumerate all admissible sizes in $O(n + m)$ time, which in turn gives all inadmissible sizes.

This can be optimized in linear time, with this simpler but equivalent algorithm. For each row i, take all maximal rectangles anchored in row i. Using the array $H_{i,j}$, it is a standard problem to compute all maximum rectangles in $O(max(n, m))$ time with stacks. If there exists a maximal rectangle of size $w \times h$, then the size $w \times (h+1)$ is inadmissible (along with anything larger than that). Regardless of the status of row $i+1$, if you repeat this for all rows, you can find all inadmissible sizes except the size $w \times 1$ case (which can be manually handled). To prove it, note that any inadmissible size you rejected in the naive solution corresponds to some maximal rectangle in the final algorithm.

Shortest solution: 1868 bytes

Problem J. Periodic Ruler

Problem idea and preparation: Dongkyu Han (queued_q)
First solver: ext71 (0:08)
Total solved team: 68

Let's store the numbers that cannot be a period of the ruler in a hash set S.

When the period is t, a pair of marks with a distance multiple of t must be of the same color. In other words, the distances of differently colored pairs cannot be multiple of t. For every such pair, we will put the divisors of the distances into S.

To do so, we have to find the divisors quickly. The maximum distance is 2×10^9, so trial division would not work. Instead, we can try dividing the distance, namely l, by positive integers less than or equal to $\sqrt{l}$. If an integer d divides l, count both d and l/d as divisors. This method counts all divisors of l and the time complexity is $O(\sqrt{l})$. Since we have to repeat this process for every pair with different colors, it takes $O(n^2\sqrt{\max l})$ in total.

Now consider a positive integer t that has not been added to S. It is possible to color the ruler such that $c_i = c_{i+t}$ for every i. However, there is one more condition for t to be a period: t must be the **minimum** positive integer that satisfies $c_i = c_{i+t}$ for every i.

To check if $t \notin S$ can be a period, i.e. no smaller period exists, we have to do the following. First calculate the colors of the marks at $0, 1, \cdots, t-1$ using the property $c_i = c_{(i \mod t)}$. If there exist marks among them that we don't know their colors, we can color them arbitrarily so that the ruler doesn't have a smaller period. On the other hand, if we know all the colors, see if some divisor d of t can be a period by checking $c_i = c_{i+d}$ for every $0 \le i < t - d$. t cannot be a period if such a divisor exists, and otherwise t can be a period. The time complexity is $O(t^2)$. (We can make it faster with $O(t\sqrt{t})$-time if we use the same algorithm to get the divisors in $O(\sqrt{t})$ time, but it is not necessary to solve the problem.)

If $t \notin S$ is larger than n, t can always be a period. It is because there must exist a mark with unknown color among the marks at $0, 1, \cdots, t-1$, since we only know the colors of $n < t$ marks. Therefore, we have to search for a smaller period only for $1 \le t \le n$. The time complexity is $O(n^3)$.

So the total time complexity to get the elements in the set S is $O(n^2\sqrt{\max l} + n^3)$. Print the size and the sum of S and it will get accepted.

Shortest solution: 1171 bytes

Problem K. Three Competitions

Problem Idea: Jaehyun Koo (koosaga)
First solver: BMSTU : (0:16)
Total solved team: 34

Build a graph of n people such that there is a vertex from i to j iff person i directly wins against j. Then this graph is a tournament graph: a directed graph which is obtained by orienting each edge of a complete graph in arbitrary direction. The given queries are reachability queries, asking whether there is a path from one vertex to another.

In a tournament graph, we can answer reachability queries in $O(1)$ with $O(n^2)$ preprocessing time. Find all the strongly connected components and order them in topological order. For given two vertices i and j, let C_i and C_j be the SCC that contains i and j respectively. Then it's easy to prove that there is a path from i to j iff $C_i = C_j$ or C_i appears earlier than C_j in the topological order.

But of course, we have to find the SCCs more quickly to solve this problem. We can do this by optimizing Kosaraju's algorithm. This algorithm computes the SCC by doing a DFS twice in a specific order. Thus, if we can quickly find a new, unvisited vertex u adjacent to a given vertex v (or report that no such u exists), then we can skip unnecessary edges and compute the SCC more quickly.

Let's focus on finding a new vertex u that has a higher rank than v in both the first and the second competition. The other two kinds of vertices can be found in the same way. We maintain a maximum segment tree, where the key is the first competition rank, and the value is a pair of (second competition rank, vertex id). Initially, all vertices are registered into the segment tree. Whenever v is visited, unregister it by putting the value $(0, 0)$. To find a new vertex, use a maximum query on $[a, n]$, where a is the rank of v in the first competition.

During the second DFS, since the graph is inverted, we have to find a new vertex that has a lower rank in at least two competitions. This can be done in the same way as above.

The time complexity is $O(n \log n + q)$, by initializing the segment trees in $O(n)$, finding a new vertex $O(n)$ times in $O(\log n)$ each, and answering each query in $O(1)$.

Shortest solution: 2094 bytes

Problem L. Utilitarianism 2

Problem idea and preparation: Jaehyun Koo (koosaga)
First solver: japan02 : Shigemura, Kawasaki, Yui Hosaka (2:28)
Total solved team: 2

1. Polynomial time algorithm. For a fixed set of agents, the maximum utility $f(S)$ corresponds to a maximum-weight matching in a bipartite graph over the edge set S. This can be determined by using minimum-cost maximum-flow (MCMF). Construct a graph with $n+m+2$ vertices, where the source connects all vaccine manufacturers, all hospitals connects to sink, and agents connects respective facilities. All edges have unit weight, and the agent edges have weight $-c$. The negation of MCMF in this flow graph corresponds to the optimal solution. (Keep in mind, that you don't have to find *maximum flow*, but just have to find *minimum cost flow.*)

As a result, the answer can be computed by $k+1$ execution of MCMF algorithm. Any standard algorithm will yield a polynomial-time solution, and any standard algorithm will time out in the given input bounds.

2. $min(n, m) + 1$ **execution of MCMF.** Consider a set of edges in any maximum weight matching. If the current agent does not belong to that matching, $f(U \setminus \{e\}) = f(U)$ trivially holds. This reduces the execution count of MCMF algorithm, but is still insufficient to solve the problem.

3. One execution, $O(min(n, m))$ **shortest path computation.** Consider a symmetric difference $D = f(U)\Delta f(U \setminus \{e\})$. As $f(U)$ and $f(U \setminus \{e\})$ forms a matching (maximum degree 1), D is a collection of paths and cycles.

If there is a cycle or a path in the symmetric difference that does not contain the edge e, then the sum of weights in their respective sets should be equal, otherwise both matching can't be optimal. As a result, we can assume that the symmetric difference is either a path or cycle that contains e.

Let's say we found the optimal solution $f(U)$ with the initial MCMF. Then the path and cycle consisting the symmetric difference corresponds to the directed path in the residual graph. The cycle case can be solved straightforwardly, since you can just find the shortest path from u to v if $e = (u, v)$. The path case requires some modification from the initial flow graph. In the end, the path containing e is also a part of big cycle that contains the source and sink. If you add a weight 0 edges from source to sink which remains unaugmented, then a shortest path from u to v represents cycle and path cases at once.

As a result, the problem can be solved with single execution of MCMF and $O(min(n, m))$ execution of shortest path algorithm. This is the intended solution, but there is an important technical detail which we have to mention.

4. Choosing the right shortest path algorithm. The shortest path algorithm in MCMF requires the handling of negative weight edges. Using Bellman-ford or SPFA requires $O((n + m)k)$ time. In most contest environment, they perform very well, to the extent that some consider them as a linear-time algorithm. But the tests are specifically designed to time out these algorithms, at least to our best effort.

In fact, there is a technique that requires at most one iteration of Bellman-Ford, usually called as *potential* *method* or *Johnson's algorithm*. This is a well-known technique which we won't discuss in detail. With this technique, we can transform each edge costs as a nonnegative number and apply Dijkstra's algorithm to find the shortest path. This yields a total $O(\min(n, m)(n + m + k) \log k)$ algorithm, which passes in time.

Shortest solution: 3429 bytes

Problem M. Yet Another Range Query Problem

Problem idea and preparation: Jongyoung Lee (Gom)
First solver: japan17 yosupo (1:32)
Total solved team: 4

Let's first consider the solution for the case $l = r$.

Let l_i be the largest j with $j < i, A_j > A_i$ (0 if it doesn't exist) and r_i be the smallest j with $j > i, A_j > A_i$ ($n + 1$ if it doesn't exist). Then, $Max(A_s, A_{s+1}, \ldots, A_e) = A_i$ if and only if $l_i < s \le i \le e < r_i$. As a result, we can decompose the 2-dimension grid into a n piece of rectangle grouped by same range maximum value. This also holds for a minimum value.

Maintain $B_{i,*}$ in a segment tree and sweep in increasing order of row number i. The insertion and the deletion correspond to a range multiplication (query type 1), and each query corresponds to a range sum (query type 2). You can use a segment tree with lazy propagation. The case $i > j$ can be handled by multiplying $B_{i,i} := B_{i,i} \times 0$ after processing the row i.

In the general case, denote the answer of the query (l, r, s, e) as $Q(l, r, s, e)$. Observe that $Q(l, r, s, e) = Q(1, r, s, e) - Q(1, l - 1, s, e)$, which means that we can assume $l = 1$.

Let's extend the above lazy segment tree to support another operation. Let S_* be an array initialized with zero. If the type 2 (sum) query is given, we compute $\sum_{i=l}^{r} S_i$ instead. If the type 3 (historic addition) query is given, we apply $S_i = S_i + B_i$ for all $1 \le i \le N$. If this query is possible, We can simultaneously maintain a sum $S_{i,j} = \sum_{k=1}^{j} B_{k,j}$ in the sweeping phase.

You can modify the lazy segment tree for these queries. For each node, store the value S_i, and the count C_i. These values denote how much time query 3 was invoked in the range but not propagated to the subtree. After processing each row, increment the value S_{root}, C_{root} by the sum of B and 1, respectively. If we encounter the node with a non-zero value C_i, we can compute how much sum should be added to their children, since their value remains untouched while the C_i was accumulated. This article might help you in understanding historic value maintenance.

In total, the problem can be solved in $O((n + q) \log n)$.

Shortest solution: 2987 bytes