Problem A. Automatic Sprayer 2

Problem idea and preparation: Suchan Park (tncks0121)

First solver: BabyPenguin (Onsite, 94 minutes), hyperbolic (Open, 61 minutes)

Total solved team: 7 (Onsite), 13 (Open)

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

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. Dynamic Short Path

Problem idea and preparation: Jaehyun Koo (koosaga)

First solver: BabyPenguin (Onsite, 253 minutes), longestpathtowf (Open, 69 minutes)

Total solved team: 1 (Onsite), 4 (Open)

For a random prime p, create a following matrix $A \in Z_p[x]^{n \times n}$:

$$A_{i,j} = \begin{cases} 1, & \text{if } i = j. \\ r_{i,j} x^{w(i,j)}, & \text{otherwise.} \end{cases} \tag{1}$$

where $r_{i,j}$ is some non-zero random element in Z_p. By Schwartz-Zippel lemma, this matrix is invertible with high probability.

Let $adj(A)$ be the adjoint (adjugate) matrix of A. By the definition,

$$adj(A)_{i,j} = \sum_{p \in P(n), p_j = i} \prod_{k \in [n] \setminus \{j\}} A_{k,p_k} \tag{2}$$

where $P(n)$ is the set of all permutations of size n.

In the graph-theoretic term, this is the product of the partition of vertices into cycles, and paths from i to j. If we construct a permutation where vertices in shortest paths are clustered as a cycle, and $p_k = k$ for all other k, we will obtain a term of degree $dist(i, j)$.

Conversely, consider some non-zero term of $adj(A)_{i,j}$ of degree k. If we remove all vertices in the cycle, we can obtain a path from i to j with length at most k. As a result, we can establish a bijection where the shortest path from i to j equals the smallest degree term in $adj(A)_{i,j}$.

In this problem, we are only interested in whether the shortest path is zero or one. Thus, we can only maintain the linear polynomial in A, which makes addition, subtraction, multiplication doable in $O(1)$. Adjoint matrix can be computed by taking an inverse and multiplying it with the determinant. Thus a single Gaussian elimination suffices. Here, we obtain a $O(Qn^3 + q)$ time algorithm, where Q is the number of updates.

To optimize this, we can use some techniques. The update of edge in graph corresponds to a single element change in a matrix A. This corresponds to a rank-1 update in A. The matrix inverse and matrix determinant can be maintained in $O(n^2)$ for each rank-1 update, by using the Sherman-Morrison-Woodbury Formula and Matrix Determinant Lemma. Using this optimization, we obtain a $O(n^3) + Qn^2 + q)$ solution.

Note that this solution is able to solve for all constant c (where $c = 2$ in this problem) with the same complexity. More specifically, we only have $O(c \log c)$ overhead for each operations. Due to this fact, the original problem had $c = 7$, but unfortunately we didn't manage to beat Floyd-Warshall by 2x factor.

Remark. We unfortunately had quite a number of unintended solution for this problem. One of them uses `std::bitset` to recompute reachability in $O(n^3/64)$ time. This solution is theoretically not bad, since all operation can be reduced to a $O(1)$ calls of matrix multiplication, which runs in $O(n^{2.376})$ time. We thought we blocked it by good margin, but the contestants were able to write much optimized solution and it eventually passed all tests.

Another one is asymptotically better, and probably much easier, than our intended solution. This solution was unknown to the problemsetter, and was only discovered in the Open Contest. The details are secret, but to give a small hint, it exploits the fact that offline queries are allowed.

Shortest solution: 5800 bytes

Problem C. Equivalent Pipelines

Problem idea and preparation: Changki Yun (TAMREF)

First solver: DO Solve (Onsite, 79 minutes), 52pwslald (Open, 49 minutes)

Total solved team: 4 (Onsite), 8 (Open)

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: 아무거나 대충 정해요 안 정해지면 그냥 이 문장 전체를 팀명으로 쓰죠 이거보다는 좋은게 나오겠지 (Onsite, 140 minutes), tlenietak (Open, 234 minutes)

Total solved team: 3 (Onsite), 1 (Open)

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: BabyPenguin (Onsite, 161 minutes), jhnan917 (Open, 158 minutes)

Total solved team: 3 (Onsite), 4 (Open)

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. Histogram Sequence 3

Problem idea and preparation: Jongseo Lee (leejseo)

First solver: BabyPenguin (Onsite, 4 minutes), khsoo01 (Open, 2 minutes)

Total solved team: 11 (Onsite), 54 (Open)

Repeatedly remove the leftmost rectangle. For example, let the histogram sequence be $B[1], B[2], \ldots, B[m]$. The leftmost rectangle would have width $B[1]$, then the next rectangle will start from $1 + B[1]$ and have width $B[1 + B[1]]$, and so on...

Shortest solution (with a little bit of code golf): 79 bytes

Problem G. Lamb's Respite

Problem Idea: Jaehyun Koo (koosaga)

Preparation: Jaeung Lee (L0TUS)

First solver: BabyPenguin (Onsite, 160 minutes), 79brue_fanclub (Open, 184 minutes)

Total solved team: 2 (Onsite), 1 (Open)

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.

With this observation, we can determine whether the champion dies or not in the interval $[1, i-1]$, as maintaining the minimum prefix, suffix, subsegment sum is an easy segment tree exercise. 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: Hexagonal Water (Onsite, 32 minutes), 79brue_fanclub (Open, 48 minutes)

Total solved team: 9 (Onsite), 13 (Open)

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 Beads

Problem idea and preparation: Hyunuk Nam (jwvg0425)

First solver: TeamakingTeam (Onsite, 15 minutes), 52pwslald (Open, 5 minutes)

Total solved team: 11 (Onsite), 24 (Open)

Suppose the number of beads currently in the bead barrel is k, the position of the leftmost cell with beads is l, and the position of the rightmost cell with beads is r.

It takes at least $r - k$ moves to gather all the beads to the left end. This is because the bead placed in the r-th cell needs to move at least $r - k$ times to make it to the k-th cell. Also, if you push the rightmost bead to the left, all the adjacent beads are pushed to the left as well, so you can gather all beads to the left end in $r - k$ moves.

By the same logic, it takes $n - l + 1 - k$ moves to gather all the beads to the right end. Since both cases must be considered, the answer is $\min(r - k, n - l + 1 - k)$.

Now, whenever beads are added and removed by a query, the l and r values need to be calculated quickly, which can be updated in $O(\log N)$ time through data structures such as std::set. So we can solve the problem in $O(N + Q \log N)$ time.

Shortest solution: 923 bytes

Problem J. Periodic Ruler

Problem idea and preparation: Dongkyu Han (queued_q)

First solver: BabyPenguin (Onsite, 36 minutes), 79brue_fanclub (Open, 65 minutes)

Total solved team: 8 (Onsite), 8 (Open)

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)

Preparation: Jaemin Choi (jh05013)

First solver: BabyPenguin (Onsite, 236 minutes), 52pwslald (Open, 252 minutes)

Total solved team: 1 (Onsite), 1 (Open)

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: N/A (Onsite), N/A (Open)

Total solved team: 0 (Onsite), 0 (Open)

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.

$min(n, m) + 1$ **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.

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.

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