Problem A. Rainbow Beads

Preparation: Jaehyun Koo (koosaga)

Subtask 1/2 (50 points)

The simplest way to check the *beautifulness* of string S can be done by simulating all three kinds of people: For non-colorblind people, check if every adjacent character is different. For colorblind people, simply replace the character `V` into `R`, `B`, and do the same. Checking this for all substring $S[i, j]$ for $1 \leq i \leq j \leq N$ gives an $O(N^3)$ algorithm, solving for subtask 2.

Subtask 3 (70 points)

We can optimize the above algorithm in the following way: Suppose that you've fixed the starting point of substring $S[i, *]$. Then, you can increase j one by one until you hit some consecutive characters which are the same for some kind of people. In this way, you can find the maximum of j which $S[i, j]$ is beautiful. This reduces the time complexity into $O(N^2)$ and solves subtask 3.

Subtask 4 (100 points)

Let's observe some properties of **beautiful** string. For red-colorblind people, the beautiful string is a repetition of red/blue character, where the red character can be violet. For convenience, let's assume the length of the string to be at least 2. Then, all blue occurrence for the red-colorblind person is actually the character `B`. For the blue-colorblind person to not see any adjacent character, any character that is adjacent to `B` should not be `V`.

Thus, we conclude the beautiful string of length at least 2 is either `RBRBRBRB...` or `BRBRBRBR....` You can find such string in linear time by identifying some pair of character `R`, `B`, extend until it hits some bad character, and reiterate from that failure point. Note that `V` is beautiful although it does not follow such patterns.

Shortest solution: 114 bytes

Problem B. Gosu

Preparation: Seunghyun Joe (ainta)

Subtask 1 (40 points)

The definition of winning path and distance can be directly replaced to path and distance in the directed graph when there is an edge from x to y iff student x wins to student y. In this graph, you can calculate $d(i, j)$ for each i and j in $O(N^3)$ time using the Floyd-Warshall algorithm, solving for subtask 1.

Subtask 2 (100 points)

It turns out we can find the Gosu in a simple way. We will prove the following lemma:

Lemma 1. *If student s has most wins among all students, then the weakness of s is less than or equal to 2.*

Proof. Suppose, weakness of student s is at least 3, then there exists t that requires distance at least 3. Thus, t wins to s, and for all student u that s can win, t also wins u, as otherwise the distance is at most 2. As a result, t wins all students that s can win, and in addition, t can win s. t have more wins than s. Contradiction. □

Therefore, you can always report that student who has most wins, and if they win to every student, weakness will be 1; otherwise 2.

Shortest solution: 145 bytes

Problem C. Voronoi Diagram Again

Preparation: Hanpil Kang (hyea)

Subtask 1, 2 (50 points)

It seems really hard to construct a Voronoi diagram directly, as you see the complex Voronoi diagram in example 2. However, It is easy to calculate whether a point X is included in a region r or not. You can simply calculate $d(X, P_i)$ for all i, and check whether $d(X, P_r)$ is minimum.
So, basically, you can draw a Voronoi diagram, check whether the region is unbounded, which means you need an infinite amount of time. But There should be less number of "important" point, right? Actually it turns out that for point $P_i = (x_i, y_i)$, checking (x_i, y_{min}), (x_i, y_{max}), (x_{min}, y_i), (x_{max}, y_i) is enough to see whether region is bounded or not, where y_{min} is minimum value among all y coordinate, x_{max} is maximum value among all x-coordinate, and so on. We will prove some lemmas.

Lemma 1. *Suppose point $X = (r, c)$ is included in region i, Then (x_i, c) is also included in region.*

Proof. Using triangle inequality and definition of region, for all j, $d((x_i, c), P_i) = |c - y_i| = |c - y_i| + |r - x_i| - |r - x_i| = d(X, P_i) - d(X, (x_i, c)) \geq d(X, P_j) - d(X, (x_i, c)) \geq d((x_i, c), P_j)$. □

In the same way, (r, c_i) is also included in the region.

Lemma 2. *It is equivalent that for any r (r, y_{min}) is in region R and, for all $c \leq y_{min}$, (r, c) is in region R.*

Proof. $d((r, c), (x_i, y_i)) = |x_i - r| + |y_i - c| = |x_i - r| + |y_i - y_{min}| + |y_{min} - c| = d((r, y_{min}), (x_i, y_i)) + |y_{min} - c|$. Change of distance is only constant $|y_{min} - c$, which does not affect comparison. □

In same way, this holds for y_{max}, x_{min}, x_{max}.
So, we can finally prove our original theorem.

Theorem 1. *If any of (x_i, y_{min}), (x_i, y_{max}), (x_{min}, y_i), (x_{max}, y_i) is included in region i, then region i is unbounded.*

Proof.
$(\Longrightarrow)$ If (x_i, y_{min}) is included in region i, than for all $c \leq y_{min}$, (x_i, c) is included in region i, which makes unbounded region.

($\Longleftarrow$) Suppose $R = |x_min| + |y_min| + |x_max| + |y_max|$. If any point $X = (r, c)$ with $d(O, X) > R$, either $|r| > |x_min| + |x_max|$, or $|c| > |y_min| + |y_max|$ (or both) holds. This means, (x_i, c) or (r, y_i) is one of 4 half-ray in lemma above, which means also one of four points also included in region i. $\square$

So, you can check this naively, in $O(N^2)$ to pass subtask 2, and naive guess checking bounding box is can pass subtask 1.

Subtask 3 (100 points)

Now, with some analysis, checking one of point is included in region i, implies that any other point do not **dominate** from any of four directions in P_i. Dominating from y_min is $x_i + y_i > x_j + y_j$ and $x_i - y_i > x_j - y_j$, and so on.
Now we can sort our point according to $x_i + y_i$, iterating, and updating the maximum value of $x_i - y_i$ to check they are dominating or not. watch out for point which have duplicate $x_i + y + i$ and $x_i - y_i$ index.
This solution can get a full score.

Shortest solution: 644 bytes

Problem D. A Plus Equal B

Preparation: Seunghyun Joe (ainta)

Subtask 1 (36 points)

We will make both A, B as a smallest power of two which is at least B.

We will increase A from 1 to $2, 4, 8, 16 \cdots$ by using operation $A+ = A$. While doing this, if B has any ones in a binary representation in a digit that A is currently on, we use operation $B+ = A$ to eliminate it. Eventually, the procedure will end with $A = B$ with at most $2 \log B$ queries.

Subtask 2 (100 points)

Observe that, $A+ = A$ is equivalent to $B/ = 2$, and vice versa, so you don't have to deal with big integers. Thus, if A is even, you can set $A/ = 2$, and if B is even, you can set $B/ = 2$, and you can safely assume both to be odd.

Without loss of generality, let's assume $A > B$. Then perform operation $A+ = B, A/ = 2$ ($A+ = B, B+ = B$). $|A - B|$ decreases as half, so this operation could only continue for $\lg N$ times. You can easily see, that you use $2 \times (1 + 2 + 3 + \cdots + 60)$ operation, which is less than 4000.

Challenge: Can you solve if $n \leq 1000$? What about $n \leq 350$?

Shortest solution: 466 bytes

Problem E. Water Knows the Answers

Preparation: Donghyun Kim (kdh9949)

Subtask 1 (18 points)

There is $2^3 = 8$ ways to choose the direction of boxes and $\frac{3!}{2} = 6$ ways to choose the order of boxes. Since $N = 3$, the water can be stored only above the middle box. Then, we can easily evaluate the amount of water storage for each arrangement.

Subtask 2 (19 points)

Since $w_i = h_i$, we don't have to consider the direction of the box. If we fixed the direction of the box, putting the 2 highest boxes at the boundary is optimal. If one of the 2 boxes was internal(not at the boundary), we can find a lower box at the boundary, and we can just swap the two boxes to get a better result. After deciding the order of the boxes, we can calculate the amount of storage in $O(N)$.

Subtask 3 (34 points)

From now, let's assume that $w_i \geq h_i$ for all boxes (initially). We can apply two steps of greedy approach for certain arrangement of boxes to make it better; First, we can move 2 highest boxes to the boundary. Second, if the boundary box has a longer width than height, set it upright. For the internal box, do it opposite. (If the box has longer height, lay it down.) This two-step always gives more (or same) water storage.

After applying those greedy steps, the candidates for the answer are much restricted: choose 2 boundary box and set them upright, then lay down all the internal boxes. If there is no internal box higher than any boundary boxes, it can be one of the candidates for the answer. If we check this condition and calculate the water storage area in $O(N)$ for each bounding box pair, it gives an $O(N^3)$ algorithm.

Subtask 4 (14 points)

First, we can calculate the area of water storage for each candidate in $O(1)$. Let $Area(i, j)$ be the area of water storage when i-th and j-th boxes are the boundary boxes. It can be calculated by following formula (Assume $w_i \geq w_j$) : $Area(i, j) = w_i h_i + w_j h_j + w_j(\sum w_k - w_i - w_j) - \sum w_k h_k$.
Each term can be evaluated in $O(1)$ after some pre-computation, so we can calculate $Area(i, j)$ in $O(1)$.

Actually, we don't have to care if there exists a higher internal box for a certain pair of boundary boxes. If there exists a higher internal box, it is definitely not the optimal arrangement and $Area(i, j)$ will be smaller than the actual amount of storage. So those $Area(i, j)$ values don't affect the final answer. So, we can just calculate the maximum value of $Area(i, j)$ for all pair of $1 \leq i < j \leq n$, which takes $O(N^2)$.

Subtask 5 (15 points)

$Area(i, j)$ can be re-written in form of $f(i) + g(j) - w_i w_j$. So, we can use convex hull trick to calculate the maximum value faster. After sorting the boxes with w_i in non-decreasing order, it can be done in $O(N)$

using stack.

Shortest solution: 716 bytes

Problem F. Eat Economically

Preparation: Gyeonggeun Kim (august14)

Subtask 1 (16 points)

There exist two solutions which solve this subtask: Dynamic Programming, and Min-cost flow. The first solution is easier, but the second solution can be extended to the full solution. We will explain the first solution here.

Let $DP_{i,j,k}$ be a minimum cost where Ho picks j lunch, k dinners from dish $1, \cdots, i$. The base case is $DP_{0,0,0} = 0$. For inductive case, you can build the recurrence by considering three cases: Using it as a lunch, as a dinner, or ignore. The answer is stored in $DP_{2K,i,i}$, and the calculation of this table takes $O(K^3)$, which solves subtask 1.

Subtask 3 (100 points)

We can model this problem with min-cost flow straightforward. There are only two types of augmenting path for lunch (and dinner vice-versa).

1. Select one for lunch among non-selected menus.
2. Change one into lunch among selected dinner menus and select one for dinner among non-selected menus.

And, for the min-cost, we must choose the shortest augmenting path among those.

When we increase the capacity of lunch and dinner one by one, then the new augmenting paths will be made. We can find the shortest path and maintain the flow network economically with the following four types of (min heap) priority queues.

1. Sorted in (*lunch*) cost among non-selected menus.
2. Sorted in (*dinner*) cost among non-selected menus.
3. Sorted in (*lunch* - *dinner*) cost among selected dinner menus.
4. Sorted in (*dinner* - *lunch*) cost among selected lunch menus.

Thus, we can solve this problem in $O(K \lg K)$ time. Choosing an inefficient data structure can give time complexity $O(K^2)$, solving for subtask 2.

In addition: When we increase capacity, the negative cycle can be made. But in this problem, we can ignore this. Why this can be?

Shortest solution: 1441 bytes

Problem G. Increasing Sequence

Preparation: Jaehyun Koo (koosaga)

Subtask 1 (22 points)

A longest increasing subsequence (LIS) that contains index i, is a concatenation of LIS that ends with index i, and starts with index i. We can compute the length of such with dynamic programming: Let DP_i be a LIS that ends with index i, then $DP_i = Max_{j<i, a_j<a_i} DP_j + 1$. Opposite direction can be done in a same way. For each i, j, you can compute the dynamic programming in $O(N^2)$ time, with index j ignored. Repeating this for $O(N^2)$ gives an $O(N^4)$ algorithm.

Subtask 2 (38 points)

You don't have to compute the DP for $O(N^2)$ time: You can simply fix the index j to remove and see whether i has its answer changed. This simple strategy gives $O(N^3)$ algorithm by computing the DP $O(n)$ time.

To further reduce this, you should do each DP in $O(N \log N)$ time. Note that $DP_i = Max_{j<i, a_j<a_i} DP_j + 1$. If you calculate the DP in increasing order of i, then the solution procedure can be rewritten as follows:

- Initialize the array $tree[*] = 0$.
- For each i, calculate the maximum among $tree[0], tree[1] \cdots tree[a_i - 1]$, and store at DP_i.
- Update $tree[a_i] = DP_i$.

This can be interpreted as a range maximum query with updates, which can be implemented in $O(\log N)$ time per query, so the DP can be calculated in $O(N \log N)$ time, resulting in $O(N^2 \log N)$ algorithm. There exists a simpler algorithm, but this method will be more helpful to analyze the full solution.

Subtask 3 (100 points)

For convenience, we will calculate the answer for all $j < i$. $j > i$ can be done in an exact same way when we reverse and negate the input.

Let S_i be the set of longest sequences among increasing subsequences end with index i. If every element in S_i has the same index j, it will decrease the maximum possible length of an increasing subsequence that contains index i when removing it. Let $J(i)$ be the largest such index j.

If every subsequence in S_i has the same index j, every element in S_i must have a prefix that is in S_j. Thus, $[J(i), J^2(i), \cdots]$ is the list we want, and if we calculate all $J(i)$, then you can count the size of such list by simple dynamic programming.

If the lexicographically smallest in S_i and the lexicographically largest in S_i has the same index j, then every element in S_i has the index j. Thus, let l_i and r_i are the smallest and the largest index j respectively such that $j < i$ and $A_j < A_i$, J_i will be LCA of l_i and r_i when we see J as parent-child relation of tree.

To calculate LCA, you can use binary lifting. The time complexity is $O(N \log N)$.

Shortest solution: 1187 bytes

Problem H. Jealous Teachers

Preparation: Seunghyun Joe (ainta)

Subtask 1 (32 points)

This subtask is a straightforward exercise of the Maximum-flow algorithms, where you can model the flow graph by adding an edge of capacity $N-1$ from source to student, and the edge of capacity N from teacher to sink. Existing edges will have infinite capacity. As the limits are small, any selection of algorithm will be accepted, for example, the Ford-Fulkerson method.

The time complexity of Ford-Fulkerson is $O(fE)$, and $f, E = O(N^2)$, thus it solves the problem in $O(N^4)$ time.

Subtask 2 (100 points)

For each subset A of students, there should be at least $|A|+1$ teachers that could receive a flower from a student in A. This is also sufficient, and it can be shown by Max-flow Min-cut argument. By Hall's theorem, there should be a perfect matching between $N-1$ students and $N-1$ teachers numbered $1, 2, .., N-1$.

With this knowledge, we construct a tree. Initialize S be an empty set and T be a set consists of one teacher N, repeat these steps:

- If T contains all N teachers then terminates;
- If there is no correct pair between students from S^C and teachers from T, then S^C have less than $|S^C|$ neighbors, so there is no solution.
- If there is a pair of $s \in S^C$ and $t \in T$, insert s in S and insert $match(s)$ in T.

If the algorithm terminated with $|T| = N$ then, for each student s, there are two distinct teacher $match(s)$ and t who was a pair with s in the algorithm.

Let's make a graph G consists of N vertices. Draw an edge between $match(s)$ and t for each student s. Then G is a tree. Once you get a tree G, you can inductively build a correct assignment of flowers, by using bottom-up recursion.

The time complexity is determined by the selection of the bipartite matching algorithm. Using the Hopcroft-Karp algorithm will result in $O(M\sqrt{N})$ time complexity.

Shortest solution: 2226 bytes

Problem I. Dijkstra Is Playing At My House

Preparation: Jaehyun Koo (koosaga)

Subtask 1 (24 points)

As there are $2N+2$ different x, y coordinates we should consider, We can do coordinate compression, and consider the plane as a grid of size $(2N+2) \times (2N+2)$. (Search for *coordinate compression* if you don't know it!) Every adjacent grid cells are connected with edges weighted by their distance. Rectangles given as an input removes edges in a certain rectangular area, which we can naively process in $O(N^2)$ time.

After doing all the compression, we are left with a weighted graph with $O(N^2)$ vertices and edges. Now, we can run the shortest path algorithm to determine the answer. As the graphs are huge, you need an efficient algorithm to solve the problem, for example, Dijkstra's algorithm implemented with priority queues.

Subtask 2 (55 points)

Our intended solution relies heavily on this core lemma:

Lemma 1. *WLOG $x_s < x_e, y_s < y_e$. An optimal path have either nondecreasing x coordinate or nondecreasing y coordinates.*

We omit the proof for a while: Actually, we don't know any proof that can't immediately solve subtask 3, so we assume that people solving subtask 2 are simply guessing the lemma to be true.

With this lemma, you can replace the shortest path algorithm with Dynamic programming, as the graph is acyclic. WLOG consider an optimal path that only has nondecreasing x coordinate. Let $DP[i][j]$ be a minimum cost to reach the grid (i, j) (assuming it's coordinate compressed). Then, the answer depends on $DP[i-1][*]$, and you can do the transition in $O(1)$ time with appropriate prefix/suffix minimums. The time complexity is $O(N^2)$.

Subtask 3 (100 points)

Now we will present the proof of the above lemma.

Proof. Let's bound the whole plane in a certain rectangle. We define a *RU-waterfall path* as follows: Move right from the starting point, and whether you hit an obstacle, you move upward until you can move right, and you continue until you escape a bounding box. Similarly, you can define RD, UR, UL waterfall paths in the same way.

Notice that RU/RD waterfall paths bound a certain area, and similarly, UR/UL waterfall paths also do. Then, we have two cases:

- Case 1. The destination is not bounded by both pair of paths. Then, take the DL-waterfall path from a destination. It will either meet RU or UR waterfall paths. If you join both, you can see that the path is the shortest possible path between two points.
- Case 2. The destination is bounded by one of them. WLOG assume it's bounded by RU/RD path pairs. One observation is, you never have to go strictly outside of that bounded area, because you will eventually go inside of the bounded area, and the border created by waterfall paths gives the shortest path toward them. This fixes the *first decision* that should be made: The path should go down until hitting by a rectangle, and then you make a binary decision of going either left or right. Simulate either of them and as soon as the destination is not bounded by RU/RD path, this reduces into Case 1. Formal proof can be done in the same way, using induction on the y coordinate.

□

Interestingly, the above proof is constructive: it gives you an algorithm to find the shortest path in exponential time and with some algorithmic techniques, a $O(n \log n)$ one. Simulate the *waterfall path* by sweep line: Start from event x_s, and if some event hit the rectangle, remove such events and create two new events in the left/right side. This simulation can be done directly in $O(n^2)$ time if you simply rule out duplicates, and it's straightforward to optimize in $O(n \log n)$ time with std::set.

The final set of events, collected in y_e, indicates whether the region was bounded by RU/RD path, and the shortest distance for arriving at some x-coordinate. If the destination was outside of the bound, then you are done. Otherwise, by the same argument in Case 1, you can just pick minimum $dist_p + |x_p - x_e|$, regardless of whether there is an actual blocking rectangle passing coordinate y_e. This runs in 260ms, but the time limit was very lenient to accommodate a possible alternative solution.

Shortest solution: 1962 bytes