Problem A. A Plus Equal B

Preparation: Seunghyun Joe (ainta)

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 continues 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: 507 bytes

Problem B. Bohemian Rhaksody

Preparation: Jaehyun Koo (koosaga)

The very first observation is, that we are essentially finding a maximum-area axis parallel rectangle. It's not hard to prove the equivalence between the "lighted area", and the axis parallel rectangle that does not contain any bulbs in interior. This observations gives something between $O(N^5)$ to $O(N^2)$, and I'll starting by describing an $O(N^2)$ solution.

We first fix the bottom side of a rectangle, and find a maximum area rectangle with the given bottom side. Note that, for a fixed *top bulb*, we have a unique maximal interval that has such bulb in a top. Then, if we fix the top bulb, we can enumerate all $O(N)$ rectangle that covers the solution. If the rectangle is not bounded by bulb but an upper bounding box edge, then you should enumerate them separately. The enumeration can be done in $O(N)$ time using a stack, and if you don't know about this, you can search about *largest rectangle in histogram* for details.

This is obviously very slow, even though TL is 10 seconds. To reduce the asymptotics, we use the *Divide and conquer* on the point set. Sort all point in x-coordinate, and we'll try to calculate maximum rectangle among the one which crosses the line $x = X_m$. To do this, we need some observation on such rectangles.

If a rectangle crosses the line $x = X_m$, it will have the left/right barrier which blocks the rectangle to expand. This barrier is either a point, or a bounding box. We split the rectangle into two parts by the line, and for each rectangle, stretch it upper and lower independently. Then, the two split rectangles are a *maximal rectangle* in each part, and for the left/right partition, we can enumerate those in $O(N)$ time. If we reverse the point of view, and find a largest rectangle induced by pair of two maximal rectangles, then they are the one with width being sum of two heights, and heights being the intersection of two interval.

Now, the "conquer" step is reduced to a line problem dealing with intervals. You are given two weighted interval set, and you want to pick one intervals from each two set, which maximizes the $(w_x + w_y) \times intersection([s_x, e_x], [s_y, e_y])$, when $[s_x, e_x]$ is the interval and w_x is the weight. However, although we have this abstraction, we still need the geometric property of this problem, so you should not forget about it.

One case is rather simple: One interval completely fits into another. Then, the length of intersection is determined by the smaller one, and you just want to find the maximum possible weight of the corresponding rectangle. If you go back to the problem, this is just finding a point in range closest to the demarcation line, which is a simple range query doable with segment trees.

Another case is the meat of this problem: Two interval has intersection but none contains the another. WLOG, assume $s_x < s_y < e_x < e_y$. You can see that the answer is $(w_x + w_y) \times (e_x - s_y)$ for this case, which is not easy to compute even if you don't have such condition.

Let's discuss about finding the maximum of $(w_x + w_y) \times (e_x - s_y)$ for arbitrary disjoint set without any limits. If you plot the points by $(e_x, -w_x), (s_y, w_y)$, then you can notice the monotonicity between the optimal matchings: For a fixed point $(e_x, -w_x)$, if it's coordinates increase, then the optimal matching's coordinate decrease. This can be proved by comparing areas of rectangles induced by each matching.

To compute the optimal matching efficiently, one can use divide and conquer optimizations or monotonic queue, and we will discuss the latter solution: For a set $(e_x, -w_x)$, and two points in set (s_y, w_y), the set can be splitted into a prefix and suffix, where the first one is more profitable than others, or vice versa. The reason why this hold directly follows from the "monotonicity", and you can use binary search to find this point. Now, you maintain a data structure which stores a point. Whenever there comes a point where picking a point with index x is a inferior choice compared to picking $x+1$ or $x-1$, then you repeatedly eliminate such point. This can be simulated using stack, which closely resembles the convex hull optimizations. Each insertions take amortized $O(\log N)$ time for binary search, thus the subproblem can be solved in $O(N \log N)$ time.

Now, the naive solution would be doing 2 nested divide-and-conquer to impose a limit on s_y, e_y. Intuitively, this corresponds to building 2D segment trees to solve the problem. Then, for each "conquer" step you have $O(n \log^2 n)$ pairs to compare, resulting in $O(n \log^3 n)$ merging solution. $T(N) = 2T(N/2) + O(N \log^3 N) = O(N \log^4 N)$, so this is too slow by any means.

Still, we will use some kind of divide-and-conquer. We will consider all point matchings where $s_y \le m < e_x$, and recurse to find another kind of point matchings. To consider all point matchings that satisfies $s_y \le m < e_x$, we just have to consider all intervals that pass through point m, and they form a "chains": Smallest interval is contained in next smallest interval, and so on, and second largest interval is contained in largest intervals. If you have this in mind, you can observe that, for each point (s_y, w_y), the matchable points in $(e_x, -w_x)$ forms a consecutive interval, and if we collect those intervals, they are monotonic: If we sort them in order of starting point, their endpoint is sorted.

Now, even with this restriction, we can still calculate the matching in $O(N \log N)$ time. To give you the idea, just suppose that every matchable intervals are having length K. Then, you can decompose the first

point set into consecutive intervals of length K, and for each query you are basically requiring an answer for prefixes and suffixes. If the matching intervals are guaranteed to be prefix/suffix of a point set, then you can slightly modify your solution, and still make it run in $O(N \log N)$ time.
Matching intervals does not have length K, but still you can find a decomposition in a very similar way: You pick one query interval, and pick a maximal contiguous intervals of interval, which all intervals have intersections, and solve the problem prefix/suffix wise. For this specific part, you can look for a model solution of very similar problem, "Mowing Mischief" from USACO FEB19 Platinum division.

Now we built this powerful black box, so we could actually fire up divide and conquer. For an interval that crosses line $y = m$, you collect them all, and run those optimal matching instance in $O(N \log N)$ time. For the recursion, if the interval does not cross $y = m$, it's either in upper or lower side so you obviously know where to insert it. If it does cross, then you have to put it either in upper, or lower side, depending on whether your point set was left or right. In total, each interval will appear in $O(\log N)$ nodes of recursion, so the optimal matching receives $O(N \log N)$ interval, resulting in $O(N \log^2 N)$ merge time. $T(N) = 2T(N/2) + O(N \log^2 N) = O(N \log^3 N)$. You can get AC, with pretty short and nice code(in a very subjective way...).

The solutions are very quickly written, so this is very sketchy and would miss some details, although I tried hard to address most issues. If you want more details, feel free to email koosaga@gmail.com, or you can write some codeforces DM. I hope to see you again in New Year Prime Contest 2019 :D

Challenge: I found a research paper which solve this in $O(N \log^2 N)$, but I didn't saw it, and I couldn't solve it.

Shortest solution: 6303 bytes

Problem C. Cactus Determinant

Preparation: Jaehyun Koo (koosaga)

Consider a graph induced by a permutation $P(i)$ (connect i with $P(i)$ for all i); if the corresponding summand is non-zero, then it must be a subgraph of the given graph G. The subgraph is a cover of G with cycles and matching edges. Observe that flipping the direction of an odd cycle changes the sign of the permutation, so these summands cancel out. We now need to count the number of covers with even cycles and a matching; the weight of a cover is $2^{\#\text{ of even cycles}}(-1)^{\#\text{ of matching edges}}$. We can compute this with a subtree dynamic programming on the "condensed" tree of cycles.

Shortest solution: 1459 bytes

Problem D. Dijkstra Is Playing At My House

Preparation: Jaehyun Koo (koosaga)

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.*

Proof. Let's bound the whole plane in a certain rectangle. We define an *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 a 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. Destination is not bounded by both pair of paths. Then, take the DL-waterfall path from destination. It will either meet RU or UR waterfall paths. If you join the both, you can see that the path is the shortest possible path between two points.
- Case 2. 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 those bounded area, because you will eventually go inside of the bounded area, and the border created by waterfall paths gives a shortest path toward them. This fixes the *first decision* that should be made: The path should go down until hit 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 same way, using induction on y coordinate.

□

Interestingly, the above proof is constructive: it gives you an algorithm to find 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 time limit was very lenient to accommodate possible alternative solution (which we didn't found).

Author note: This is the most beautiful solution I've ever seen in year 2019. I hope you had felt the same excitement and joy as much as I did!

Shortest solution: 1962 bytes

Problem E. Eat Economically

Preparation: Gyeonggeun Kim (august14)

If we ignore time complexity, we can model this problem with min-cost flow straightforward. There are only two types of the 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 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 $O(N \lg N)$ time.
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 F. Fruit Tree

Preparation: Younghun Roh (Diuven)

Let us describe the method for finding the majority in a set. We maintain a pair (x, c), where x is a current candidate for being a majority and the meaning of c will be explained later.

When the new value y is added, the pair should be updated as following:

- if $x = y$, the new pair is $(x, c + 1)$;
- if $x \neq y$ and $c > 0$, the new pair is $(x, c - 1)$;
- otherwise the new pair is $(y, 1)$.

Finally we get some (x, c) pair. Now either x is the majority or there is no majority at all.

Brief explanation: we "match" pairs of distinct numbers and remove them from the array. If there is a majority then it will be the only element to survive the process. (This algorithm is known as *Boyer-Moore majority algorithm*, so one who needs detailed informations could search it on Google.)

Note that we can not only add a single element to the pair but merge two pairs completely. Let two pairs be (x_1, c_1) and (x_2, c_2). WLOG $c_1 \geq c_2$.

- if $x_1 = x_2$, the new pair is $(x_1, c_1 + c_2)$;
- otherwise the new pair is $(x_1, c_1 - c_2)$.

To solve the problem on a tree, we build a binary lifting structure. For each vertex v and each $0 \le k \le \log n$ we store the outcome of the algorithm as if it would be run on all vertices on a vertical path of length 2^k going up from v. Using this information we may find a candidate for majority for each query from the input. Finally we need to calculate the frequency of each candidate on a corresponding path. It may be done offline using a single DFS where we store the number of occurrences of each element on a path from the root to the vertex being traversed and calculate the number of occurrences for the query in its endpoints and their LCA.

Shortest solution: 2038 bytes

Problem G. Good Set

Preparation: Seunghyun Joe (ainta)
Let each number from 0 to $2^k - 1$ correspond to a subset of $\{0, 1, 2..., k-1\}$.
If a good set contains an empty set and $\{0, 1, 2..., k-1\}$, then the set is a topology of $\{0, 1, 2..., k-1\}$
There is a bijection between topologies of $\{0, 1, 2..., k-1\}$ and a transitive digraph consists of k vertex.
Shortest solution: 1480 bytes

Problem H. Hard To Explain

Preparation: Taekyu Kang (imeimi)

Our intended solution was to build persistent segment tree with non-amortized CHT for each node, but our tester pointed out that this is an problem that can be very straightforwardly solved by persistent Li-chao segment in $O((N+Q)\lg^2 N)$ time. We have no idea what is Li-chao segment tree, neither we have time to elaborate our solution, so sadly we'll leave this section to leaders. Btw, both solutions are online.

Shortest solution: 1449 bytes

Problem I. Increasing Sequence

Preparation: Jaehyun Koo (koosaga)

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 counting the size of this is easy when we know about J.

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.

Shortest solution: 1187 bytes

Problem J. Jealous Teachers

Preparation: Seunghyun Joe (ainta)

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.

Shortest solution: 2226 bytes