Problem A. NJU Emulator

First, we consider the idea of doubling, for any $N \geq 2$, we can obtain N using $(\times 2)$ or $(\times 2 + 1)$ from 1 in $\lfloor \log_2 N \rfloor$ steps. Then we may come up with an algorithm to construct any N within $O(\log N)$ s86 instructions as follows:

Use 3 instructions "p1, dup, add 1"to push 1 and 2 into stack, then use "p1"and $(\times 2)$ or $(\times 2 + 1)$ for $\lfloor \log_2 N \rfloor$ times to get N.

It takes at most $2\lfloor \log_2 N \rfloor + 5$ instructions to construct N, which is too large. To improve our algorithm, we can consider applying the method of four Russians, i.e., try to use a larger base k instead of the base 2.

For any $k \geq 2$ and $N \geq 2$, we can obtain N using $(\times k + p)(0 \leq p < k)$ from some $x(1 \leq x < k)$ by $\lfloor \log_k N \rfloor$ steps. Thus we can use $2k - 1$ instructions "p1, dup, add 1, dup, add 1, ..."to push $1, 2, ..., k$ into the stack, then use "p1, add (x-1)"and $\lfloor \log_k N \rfloor$ times of $(\times k + p)(0 \leq q < k)$ operation to get N.

It takes at most $2\lfloor \log_k N \rfloor + 2k + 2$ instructions to construct N. Take $k = 7$(or $8, 9, 10, 11, 12$), we obtain an algorithm to construct any $N < 2^{64}$ within 60 s86 instructions, which is close to our goal.

For further improvements, note that for a base k, if we have obtained $1, 2, ..., \lfloor k/2 \rfloor$ and k in the stack, then for any $x \leq k$, we can get x in 2 steps as following:

- "p1, add $x - 1$"when $x \leq \lfloor k/2 \rfloor$.
- "dup, sub $k - x$"when $x > \lfloor k/2 \rfloor$.

Then for $N > k$, we can obtain N using $(\times k + p)(0 \leq p \leq \lfloor k/2 \rfloor)$ or $(\times k - p)(1 \leq p \leq \lfloor k/2 \rfloor)$ from some $x(1 \leq x \leq k)$ in $\lfloor \log_k N \rfloor$ steps. Thus we can use $k + 1$ instructions to push $1, 2, ..., \lfloor k/2 \rfloor$ and k into the stack, then construct x in 2 steps, finally applying $\lfloor \log_k N \rfloor$ times of $(\times k + p)(0 \leq p \leq \lfloor k/2 \rfloor)$ or $(\times k - p)(1 \leq p \leq \lfloor k/2 \rfloor)$ to get N.

It takes at most $2\lfloor \log_k N \rfloor + k + 4$ instructions to construct N. Take $k = 16$(or $12, 14$), we can obtain an algorithm to construct any $N < 2^{64}$ within 50 s86 instructions.

Problem B. Just another board game

Consider what is the optimal strategy if the "instant-ending"operation is not allowed. We can reinterpret the operations of the two players as follows:

> We start with $r = 1$ and $c = 1$. In each turn, if it's Roundgod's turn to move, he can set c to some arbitrary value satisfying $1 \leq c \leq m$. If it's kimoyami's turn to move, he can set r to some arbitrary value satisfying $1 \leq r \leq n$. After k rounds, the game ends, and the value of the game is $a_{r,c}$.

We consider the process backward. If it's Roundgod to move in the last turn, he should obviously choose c to be the position of the maximum element in the current row, and thus kimoyami, who is second-to-last to move, should choose r to be the row with the minimal maximum. Similarly, if it's kimoyami to move in the last turn, he should choose r to be the position of the minimum element in the current column, and thus Roundgod, who is second-to-last to move, should choose c to be the row with the maximal minimum.

There still exists a special case where $k = 1$, i.e., the game only lasts for at most one turn, then clearly it's optimal for Roundgod to choose the maximum in the first row.

So, let's now consider the case where the "instant-ending"operation is allowed. I claim that this operation wouldn't make much difference to the strategy for both players, except for the first move of Roundgod.

Why is this the case? Because if you choose the second operation after your opponent has moved, it is never better than doing nothing(keeping the chess unmoved), as in the optimal strategy of your opponent, he would also do nothing(or if he chooses the second operation, it's still the same), otherwise, if he chooses another r(or c) different than his last move, which is clearly not optimal, and thus you can still have the same value without doing the second operation.

To summarize, the answer can be calculated as follows:

Let $rowmax_i$ be the maximum element in the ith row, and $colmin_i$ be the minimum element in the ith column.

- If $k = 1$, then the answer is $rowmax_1$
- Otherwise, if k is odd, then the answer is $max(a_{1,1}, \min_{i=1}^{n} rowmax_i)$
- Otherwise, k is even, and the answer is $max(a_{1,1}, \max_{i=1}^{m} colmin_i)$

The overall time complexity is $O(nm)$.

Problem C. Dota2 Pro Circuit

First, we show how to compute $worst_i$ for each team, and $best_i$ for each team can be computed in a similar manner.

For fixed i and k, in order to check whether there exists a plan that k teams have a higher score than team i, we will let team i scores b_n in the tournament, and the k teams with the initial highest score (exclude team i) getting $b_1, b_2, \ldots, b_k$ scores in the tournament. We will match the initial scores and tournament scores in a greedy way: the team with the highest initial score gets b_k scores in the tournament and the team with the second-highest initial score get b_{k-1} scores in the tournament, and so on. Then we check if all selected k teams' final scores are strictly higher than team i. This takes $O(n)$ time.

In order to compute the answer for all i, using binary search would take $O(n^2 \log n)$ time, which would be too slow. An obvious observation is that, if we sort the teams by their initial scores, then $worst_i$ is non-increasing. Thus we can use the method of two pointers to achieve a total complexity of $O(n^2)$.

Problem D. Into the Woods

For a point $P(x, y)$, the Manhattan distance between P and $O(0, 0)$ is $|x| + |y| = \max\{|x + y|, |x - y|\}$. Assume Roundgod walks through a path $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Let A be the maximum Manhattan distance between any point of the path to $(0, 0)$, then $A = max\{\max_{i=1}^{n} |x_i + y_i|, \max_{i=1}^{n} |x_i - y_i|\}$.

By the linearity of expectation, what we need to calculate is

$$E(A) = \sum_{k=1}^{n} Pr[A \geq k] = n - \sum_{k=1}^{n} Pr[A \leq k]$$

.

Here's a very useful(and beautiful) trick for this kind of two-dimensional random walk: Define random variables $a = x+y$ and $b = x-y$, then a and b are independent random variables with the same distribution! Formally, assume Roundgod's position is (x, y). In each step, $x + y$ and $x - y$ will independently increase or decrease by 1 with $\frac{1}{2}$ probability.

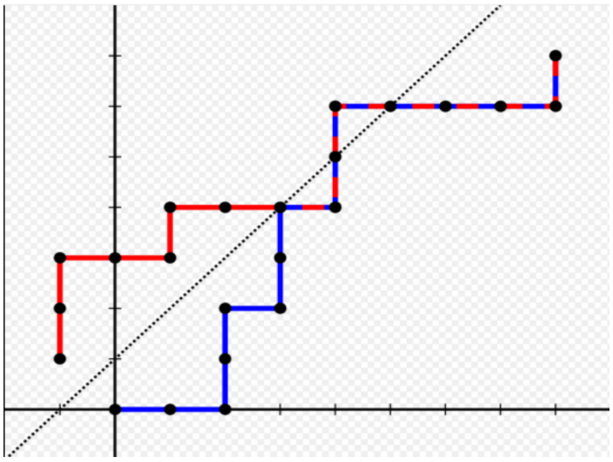
Now the problem is reduced as follows: We go on a one-dimensional random walk on the x-axis for n steps, starting from $x = 0$, each step randomly setting $x = x + 1$ or $x = x - 1$ with probability $\frac{1}{2}$ each. Let p_k denote the probability that the path of the random walk is completely inside interval $[-k, k]$. Compute p_k for each $k = 1, 2, \ldots, n - 1$. Then the answer is

$$E(A) = n - \sum_{k=1}^{n-1} p_k^2$$

.

We can now transform this to an equivalent problem: suppose one starts from $(0,0)$, walks for n steps, with each step only allowed to go right or go up. Let c_k be the number of paths which don't collide with line $y = x + (k+1)$ or the line $y = x - (k+1)$, then $p_k = \frac{c_k}{2^n}$. To compute this number, one still need to solve another subproblem: Given a point P, line $l_0 : y = x + a$ and line $l_1 : y = x - b$, how many paths from $(0,0)$ to P are there that don't trespass line l_0 or line l_1? One may recall a similar but simpler problem(also known as a generalization Bertrand's ballot theorem): Given a point P and a line $l : y = x + a$, how many paths from $(0,0)$ to P are there that doesn't trespass line l? Note that if $a = 0$, then this number is known to be the Catalan number.

So now we finally reveal the key idea to this problem, which I think is strikingly beautiful. It's called André's reflection principle, whose main idea is captured by the following picture.



By the reflection principle, we can see that the answer when only one restricting line exists is the difference between two binomial coefficients. And by an inclusive and exclusive version of reflection principle, let $f(P)$ be the mirror reflection point of P and line l_0, $g(P)$ be the mirror reflection point of P and line l_1, $C(P)$ be the number of ways of going to point P starting from $(0,0)$, then the answer is

$$C(P) - C(f(P)) - C(g(P)) + C(g(f(P))) + C(f(g(P))) - \dots$$

.

Thus we can compute c_k as

$$c_k = \sum_{i=L}^{R} \binom{n}{i} + 2\sum_{j\geq 1}(-1)^j \sum_{i=L}^{R} \binom{n}{i-(k+1)j},$$

where $L = \lceil \frac{n-k}{2} \rceil$ and $R = \lfloor \frac{n-k}{2} \rfloor$.
If we preprocess the prefix sum of $\binom{n}{i}$, then c_k can be compute in $O(\frac{n}{k})$ time, and thus the overall time complexity is $\sum_{i=1}^{n} \frac{n}{i} = O(n \log n)$.

Problem E. Did I miss the lethal?

This problem can be reinterpreted as a game:

> There are n cards in your hand, and the ith card has corresponding d_i and a_i on it. You and an opponent(the dealer, that's what we call it in Hearthstone) take turns to move in each round:
>
> 1. You choose a card with d_i and a_i on it, remove the card and deal d_i damage.
> 2. The opponent chooses a_i cards and removes them.

You want to maximize the total damages you deal while the opponent wants to minimize it, you should compute the answer if you and the opponent play optimally.

Note the that each a_i is in $\{1, 2, 3, 4\}$, thus we can divide all n cards into 4 piles: all cards with $d_i = j$ is split into the jth pile ($1 \leq j \leq 4$). Let n_1, n_2, n_3, n_4 denote the number of cards in each pile initially.

Clearly, a greedy strategy follows: Once you or the opponent decide to remove a card from a pile, the card with the highest damage in that pile is chosen. Therefore for each pile, we can sort all cards by their d_i, and we can solve the problem using a dynamic programming approach:

Let $dp[x_1][x_2][x_3][x_4][0]$ denote the answer where the jth pile remains x_j cards for each $1 \leq j \leq 4$ and it is your turn to choose a card. Let $dp[x_1][x_2][x_3][x_4][k](k = 1, 2, 3, 4)$ denote the answer where the jth pile remains x_j cards for each $1 \leq j \leq 4$ and it is the opponent's turn to choose k cards. The state transitions can be computed in $O(1)$.

The total number of states is $5n_1n_2n_3n_4$, where $\sum_{i=1}^{4} n_i \leq 200$, Thus the number of states reaches the maximum when $n1 = n2 = n3 = n4 = 50$, which is $50^4 * 5$, and should easily fit into time limit.

Problem F. Guess The Weight

First, consider the optimal strategy for uuzlovetree, which is easy: after drawing the first card, he should look at the remaining cards in the deck and choose "Small"or "Large"based on which side has a greater number of cards compared to the first card he drew is.

A crucial observation is that, for any deck, there exists some "midpoint"mp, such that in the optimal strategy, uuzlovetree should choose "Large"if the first card he drew has a number less than or equal to mp, and should choose "Small"if the first card he drew has a number greater than mp.

Assume that the current multiset of cards is SC, and the number of cards in the deck is s(i.e.,$|SC| = s$). Also we define $L = \{x | x \in SC \wedge x \leq mp\}, R = \{x | x \in SC \wedge x > mp\}$ then we can write the probability of drawing the second card as

$$Pr[DrawSecond] = 1 - \frac{\sum_{x \in L, y \in L} [x \geq y] + \sum_{x \in R, y \in R} [x \leq y]}{s(s+1)},$$

by considering all possible combination of the first two cards on the top of the deck.

Thus, by maintaining the number of cards and the number of pairs of cards with different values in each interval in a segment tree, we can calculate this probability in $O(\log \max a_i)$. Also, the midpoint can be found in this time using a descent on the segment tree. The overall time complexity is $O(n + \max a_i + q \log \max a_i)$.

Problem G. Boring data structure problem

Consider maintaining the left side(left to the element in the middle) and the right side(right to the element in the middle) by two doubly-linked lists L and R, so that the first two insertion operations can both be done in $O(1)$ time per query.

After an operation has been done, adjust the sizes of the two doubly linked lists as follows:

- If $L > R$, move the last element of L to the beginning of R
- If $L < R + 1$, move the first element of R to the end of L

This adjustment clearly has $O(1)$ amortized complexity per query.

When deleting the element, we only need to mark the element as deleted and decrease the size of the corresponding doubly linked list by one. This is known as a 'lazy' operation, we can propagate the deletion only when we are asked to perform operations(move or query) on it, which makes deletion also amortized $O(1)$ complexity.

Therefore, the overall time complexity is $O(n+q)$.

Problem H. Integers Have Friends 2.0

First, observe that if we take $m=2$ and either taking all odd a_i or all even a_i, we are guaranteed to have an answer at least $\lceil \frac{n}{2} \rceil$.

Randomly choose two distinct indices $1 \le x < y \le n$, the probability that the optimal answer contains both of them is at least $\frac{1}{4}$. If they are both contained in the answer, then m must be a factor of $|a_x - a_y$. In fact, we only need to check those prime factors of $|a_x - a_y$. Every positive x satisfying $x \le 4 \times 10^{12}$ has at most 11 distinct prime factors, and finding all of them can be easily done in $O(\sqrt{\max a_i})$ time if precomputing all prime numbers up to 2×10^6.

Repeat the process above for K times. We will err with probability at most $(\frac{3}{4})^K$. The running time will be $O(K\sqrt{\max a_i} + 11Kn)$, taking $K=40$ should both has a negligible probability of error and easily fits into the time limit.

Problem I. Little Prince and the garden of roses

Clearly, we can solve the problem for each color of the roses independently. Fix a certain color c. We need to assign colors to (the stems) of roses with color c. A common reduction in the grid works as follows: Consider the $n \times n$ grid as a bipartite graph $G=(V=L \cup R, E)$ with $|L|=|R|=n$, and consider each rose at position (u,v) as an edge between the uth vertex in the left and the vth vertex in the right part. One can find out this actually becomes the problem of edge coloring of bipartite graphs(i.e., give each edge a color, such that no two edges incident to the same vertex have the same color, minimize the number of distinct colors used).

So how many colors do we need? Let Δ be the maximum degree of a vertex in the bipartite graph, then obviously we need at least Δ colors. We can show that Δ colors are also sufficient for bipartite graphs(Vizing's Theorem: The edge chromatic number of a general graph is Δ or $\Delta+1$). There are two possible approaches:

First, we show a constructive algorithm that works in $O(|E||V|)$. Let's color the edges one by one in some order. Let (x,y) be the current edge. If there exists color c that is free in vertex x and vertex y then we can simply color (x,y) with c. If there is no such color then there are a couple of colors c_1, c_2 so that c_1 is in x and not in y, while c_2 is in y but not in x. Let's make vertex y free from color c_2. Denote z the other end of edge from y with color c_2. If z is free from color c_1 then we can color x, y with c_2 and recolor y, z with c_1. So we make an alternation. If z is not free from color, c_1 let's denote w as the other end of the edge from z with color c_1. If w is free from color c_2, then again, we can do alternation. We will eventually find an alternating chain because the graph is bipartite. To find the chain, we can use a depth-first search, and each chain contains no more than n vertices, so the overall complexity is $O(|E||V|)$.

Next, we show an easier-to-understand(but slower) algorithm. We first add dummy edges between vertices in L and R(maybe with multiplicity) so that every vertex in L and R has degree Δ. Then we find any perfect matching, color them with the same color, and erase the matching edges, repeating for Δ times. It will also yield an edge coloring with Δ different colors. The correctness of this algorithm can be proved through Hall's theorem, which we will omit here. The time complexity is $O(\Delta|E|\sqrt{|V|})$, which can get accepted if implemented well.

The overall time complexity, translating concerning n, is $O(n^3)$ for the first approach and $O(n^{3.5})$ for the second approach.

Problem J. Unfair contest

After sorting $a_1, a_2, \ldots, a_{n-1}(b_1, b_2, \ldots, b_{n-1})$, the newly added $a_n(b_n)$ may fall into three categories, i.e., among the s highest scores, among the t lowest scores, or neither. For each of these categories we have a valid possible range of $a_n(b_n)$, the remaining is just careful and thorough case analysis. Be careful with the case $s = 0$ or $t = 0$. The time complexity is $O(n \log n)$, due to sorting.

Problem K. ZYB's kingdom

First, let's consider how to solve the problem assuming $k = 1$ for each query. The event 1 is equivalent to, for each subtree separated by v, let s be the sum of c in this subtree, then for each vertex u in the subtree, we add s to then subtract c_u from its income.

There exists a folklore $O(q\sqrt{n} \log n)$ algorithm by considering vertices with degree $\leq \sqrt{n}$ and $\geq \sqrt{n}$ separately and using segment tree to update the answer. However, a much more clever approach can solve this in $O(q \log n)$. The idea is to first root the tree at any vertex then do a heavy-light decomposition on the tree. Then for each event of type 1, we only update the subtree that is connected to the father of v and connected to the only heavy child of v. This takes $O(\log n)$ time as we only update two subtrees. For each event of type 2, after querying the answer of v in the segment tree, what are we missing? We need to add the contribution of updates of u such that u is an ancestor of v, but v is not in the subtree connected to the heavy child of u. Due to the property of heavy-light decomposition, there are at most $O(\log n)$ such vertices, so by saving the updates in an array when encountering event of type 1, and iteratively go up along heavy chains when encountering event of type 2, we now can solve the problem in $O(\log n)$ per event, so the overall complexity is $O(q \log n)$.

Then what happens if $k > 1$? Everything is basically the same, you just need to apply the same idea to each component. Here one may need to build a tree structure(a.k.a, virtual tree)with respect to the k forbidden vertices, then solve recursively on this tree structure. The overall complexity is $O(\sum k \log n)$.