A Square Jutsu!

Author: Yu

Observe that for any cell at position (i, j), all cells (i', j') where $i' \leq i$ and $j' \leq j$ will have the property $c_{i',j'} \leq c_{i,j}$. Similarly, for any cell at position (i, j), all cells (i', j') where $i' \geq i$ and $j' \geq j$ will have the property $c_{i',j'} \geq c_{i,j}$. Thus, for each query, we can simply search for the top-left and bottom-right corners of the largest square.

We can start by finding a suitable top-left corner for every column in the grid. For column j, binary search downwards until we find the first i where $a \leq c_{i,j} \leq b$. We can use this point as our top-left corner because any $i' > i$ will just yield a smaller square.

Once we have found our top left corner at position (i, j), we can binary search diagonally to find the largest k where $a \leq c_{i+k,j+k} \leq b$ and $\max(i + k, j + k) \leq N$. The area of this square is k^2.

The maximum area found over all columns will be the answer for our query. We repeat this process for each query. Thus, our runtime turns out to be $O(Q \times N \times 2\log(N))$, which is sufficient for $N \leq 10^2$ and $Q \leq 10^4$.

B Angel Beats

Author: oops

Naïve Solution. Each query can be solved naïvely using DP. Let $f(i, j)$ be the number of ways to select angels from groups $1 \ldots i$ such that their sum is equal to $j \bmod 2^m$, and let S_i denote group i. The recurrence can be written as

$$f(i, j) = \sum_{a \in S_i} F(i - 1, j - a \bmod m)$$

DP Uncomputation. To speed up the solution, we don't actually need to recompute the entire DP table at each step. In particular, you can view $f(i, \cdot)$ and $f(i - 1, \cdot)$ as vectors, and the recurrence sum as a matrix-vector product. By inverting the matrix, we can essentially rollback the DP, removing group i from consideration. We can then compute the recurrence based on the modified group, S'_i, adding it back in. Also, since it does not matter what order we process the groups in, this implies that these matrices commute; in other words, we can do this trick for any i on the final DP table, without needing to also rollback $i+1, \ldots, n$.

FFT. The above solution is still too slow for large m. However, we can view these not as matrices, but as polynomials mod $x^{2^m} - 1$, and we can utilize FFT to compute these products faster. However, using FFT requires a conversion to and from the DFT representation, and a modular reduction step to ensure that the final polynomial still has degree at most $2^d - 1$. However, in this case, it is unnecessary. Since $a^{2^d} \equiv 1$ for any a, using only 2^dth roots of unity will naturally compute the correct evaluations for the reduced polynomial. Moreover, DFT can be used additively to easily remove and insert a new term into the polynomial. (You should compute the DFT for this single term naively, since it will only take $O(2^m)$ time.) Constructing a single term of the answer naively from a DFT is also faster, so you should use this same trick.

Note: the matrix trick only works if the matrices are invertible, which is not the case here. (The polynomials are also not invertible.) However, doing everything in terms of DFT by-passes non-invertibility since the point-wise representations are invertible (almost, you need to store products as a tuple of non-zero product and number of zeros, in order to properly handle division by 0).

C Attack on Titans

Author: Prayaag

D No Game No Life

Author: Dylan
These first steps aren't necessary but might help with understanding the solution. Since each c_i corresponds to a reduction from the score, lets just negate these and say that we *add* negative values to the score. Let's now negate all c_i *and* a_i, and instead of finding the minimum possible score, we will look for the maximum, and then negate that to get the answer. So at this point we've just negated all a_i, not c_i.

Now we will notice a few things. First, if a character c is in the set, we remove every occurrence of c from the string (and replace it with a dot), so the cost of including c in the set of *remaining* characters will be the sum of all a_i in positions of the string which contain character c. (Remember we are trying to maximize the score, so each a_i can be seen as costs which we want to minimize.) Second, for every string r_i, it's value c_i can only be added to the score (and must be added) if all characters it contains are not in S (i.e. they remain in the string after the operation is performed). So in order to add c_i to the score, we must keep all the letters in r_i in the string, so in a sense the inclusion of r_i depends on the inclusion of all the characters it contains, and the value we get from including it is equal to c_i multiplied by the number of occurrences of r_i in the string.

Since the string s can be large, and so can each r_i, the naive way of checking how many times r_i occurs in s would be too slow for this problem. To speed this up, we can use a polynomial hash to check for each possible position of r_i in s if it occurs there in $O(1)$ time, which is fast enough.

Now, this problem is equivalent to the project selection problem, where each r_i is a project, each character in $\{a-z, A-Z\}$ is a machine, and there is a dependency between a project and a machine if r_i contains c. To solve the project selection problem, we can construct a graph with $1+52+M+1$ nodes, and find the minimum cut of the graph. The first node is the source, the next 52 represent machines, the next M nodes are projects, and the last node is the sink. If there is a dependency between a project and machine we add an edge with infinite capacity from the machine to the project. For each machine we add an edge from the source to its corresponding node with capacity equal to the cost of the machine (the sum of a_i at all positions with character c). We also add an edge from each project node to the sink with capacity equal to that of the value of the project (c_i multiplied by the number of times it occurs in s). Now we can find the minimum cut using any flow algorithm. The value of the flow is the maximum sum of values - costs, so we can negate this and to get the answer to the original problem. To find the string itself we only include characters from the original string if their corresponding node in the graph is not connected to the source node after performing the flow algorithm, and we put dots otherwise.

E Monster Slayer

Author: Adithya (abhonsley00)

F Karuta Memories

Author: Aaryan

First, suppose there are no updates. For a leaf that detaches at time t at height y, it will fall at times $t, t+1, \ldots, t+y-1$. Therefore, the x position the leaf ends up at will be $(a_t + \cdots a_{t+y-1}) + k \cdot (d_t + \cdots d_{t+y-1})$. In other words, we can represent each leaf with a line. To find which line is at a maximum at a certain point k, we can maintain an upper hull of lines, similar to the idea behind convex hull trick. We can then binary

search on the vertices of the upper hull to find which line contains the answer, and just evaluate that line at k.

To support updates, we can use square-root decomposition. First, since $a_i, d_i > 0$, a leaf whose interval completely contains another leaf's interval will always have a higher position than that leaf. We can first do some preprocessing to remove leaves that are completely contained within other leaves and sort the rest of the leaves in order.

Each leaf is then placed into a bucket. If we are updating a_i or d_i and every leaf in the bucket contains that time, then every leaf is shifted by the update. Therefore, we can lazily update a counter for the entire block instead of updating the entire block.

If only some of the leaves in the bucket contain that time, then we can recompute the upper hull for the entire bucket. Since each of the buckets are in sorted order, at most two buckets will be recomputed, so this is overall still fast. To query, we can query the upper hull for each of the buckets. The time complexity is $O(n + q\sqrt{n}\log(n))$ if we choose a block size $B = \sqrt{n}$, but this can be improved by picking a smaller block size.

G Howl's Moving Castle

Author: Jimmy

Since the castle can be represented as a tree, that means there is a unique path from a_t to b_t. Let's first determine what it means for there to be an unobstructed path between a_t and b_t (in other words, a_t and b_t are in the same partition).

Lowest Common Ancestor Finding this path comes down to finding the lowest common ancestor of a_t, b_t. There are a few ways to do this in $O(\log n)$ per query, such as binary lifting or reducing to range minimum queries, and any of them work. See: finding LCA

Now let's try to determine which hallways must be unobstructed. For any task defined by a path $a_t \to b_t$, we know that every edge along the path must be unobstructed, i.e. a_t, b_t must be in the same subset. Then for every task, we can go through and mark all the required edges.

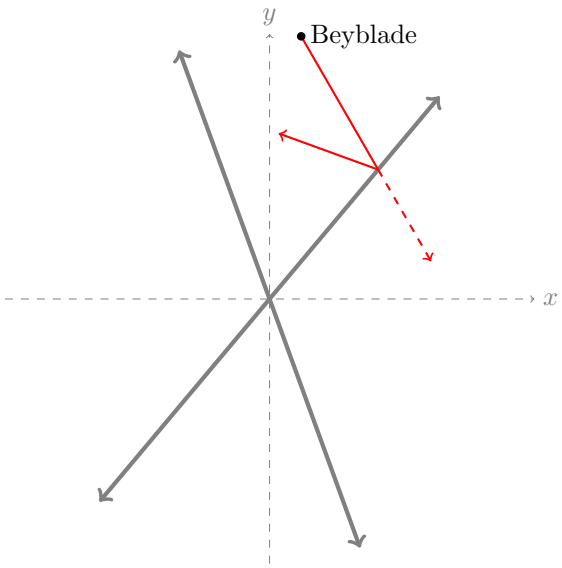
Partition Count We can then count the number of required edges r, and therefore also the number of non-required edges $N - 1 - r$. We argue that the number of possible partitions is 2^{N-1-r}. To see why this is the case, try taking some arbitrary non-required edge $e = (u, v)$. For every possible partition where e is present (i.e. u, v are in the same subset), there is a matching partition where e is absent (u, v are not in the same subset, all other things being equal).

Removing Redundancies An efficient LCA algorithm runs in $O(\log n)$. However, we also need to find the path, which takes $O(N)$ in the worst case (the path goes through every room). So this runs in $O(TN)$, which is unfortunately not efficient enough. We can improve this by noting that we often end up marking the same edge many times. So instead of iterating over the path for every task, let's simply mark where each path starts and ends. Specifically: for a path between u, v going through LCA l, we increment a counter at u, v, and decrement the counter by 2 at l. Then we can run a single depth-first search after processing all queries. We keep a running sum of the counter s (from the leaves upward), which tells us that at any given point, s paths use the current edge. Specifically, if $s = 0$, then the edge is not required.

This approach runs in $O(N + T \log N)$, which is much better.

H Beyblade Battle

Author: Aditya



Distance The first thing to note is that, as far as distance is concerned, the walls are completely irrelevant. Indeed, as shown in the image above, instead of reflecting the trajectory of the beyblade off of a wall that it collides with, one can imagine "reflecting the world", so that the beyblade continues to move in a straight line. Reflections about lines passing through the origin preserve distances to the origin, so this means that we simply need to calculate the distance between the initial straight-line trajectory of the beyblade and the origin. There are two cases.

1. If the beyblade is moving away from the origin to begin with, the distance is the initial distance $\sqrt{x^2+y^2}$.
2. Otherwise, we use the point-line distance formula. For a line $Ax+By+C=0$, the distance to a point (x_0, y_0) is
$$\frac{|Ax_0+By_0+C|}{\sqrt{A^2+B^2}}$$
Our trajectory has slope $\sin(\theta)/\cos(\theta)$ and passes through (p_x, p_y), and so can be written as
$$-\cos(\theta)x+\sin(\theta)y+(\cos(\theta)p_x-\sin(\theta)p_y)=0.$$
Plugging this into our distance formula gives $|\cos(\theta)p_x-\sin(\theta)p_y|$

We can determine which case we are in by checking the sign of the dot product of the position vector $p=(p_x,p_y)$ and the trajectory vector $t=(\cos(\theta),\sin(\theta))$.

Number of bounces The key observation for this part is that the angle of incidence increases by a fixed amount every bounce, until it hits 180 or more, at which point there will be no more bounces. You can convince yourself of this fact by either using the above observation of "reflecting the world", or just by doing some angle-chasing. So, if you can calculate the angle of incidence and this fixed amount, you can calculate the number of bounces by dividing the right two values.

First, the amount that the angle of incidence increases by (or equivalently, the amount that 180 minus the angle of incidence *decreases* by) is equal to the angle formed by the walls. However, this angle depends on which side of either wall our beyblade is. Let the angle between our initial position vector (p_x,p_y) and the positive x-axis be γ. Knowing that $\alpha<\beta$, we get that if $\alpha<\gamma<\beta$ or $180+\alpha<\gamma<180+\beta$, the angle between the walls is $\beta-\alpha$, and otherwise it is $180-(\beta-\alpha)$. Call this value δ.

Now we need to compute 180 minus angle of incidence. To calculate this, we need to first check which wall we hit first. One way of doing this is to compute the ray-line intersection of our initial trajectory and our two walls, and see which intersection time is sooner. It's important to be careful here and check for parallel lines to avoid division-by-zero errors.

Now, we know which wall we hit first. We take the difference of θ and one of $\alpha, \beta, 180+\alpha, 180+\beta$, depending on which side of which wall we are hitting first. We can use this calculation to determine 180 minus the angle of incidence, call this ϕ.

Finally, we can take ratio of this number and δ, and take the ceiling to get our final answer.

One important note is that, if $\phi = k\delta$ for some integer k, then calculating $\lceil \phi/\delta \rceil$ can give the wrong answer if ϕ and δ are stored as floating point numbers due to precision issues. Therefore, everything needs to be stored losslessly in some way. The input specifies that each angle is given as a fixed point number with 5 values after the decimal point, which allows one to store these values as fractions with a denominator of 10^5. So, one way of approaching this issue is to use/create a Rational class that handles these values exactly as ratios of integers.

I Snack Time

Author: Zeki

Notation $[n] = \{1, 2, \ldots, n\}$ for $n \in \mathbb{N} \setminus \{0\}$.

Number Theory Suppose that the nodes along a $u \rightsquigarrow v$ path are $u = u_1, u_2, \ldots, u_k = v$. Then suppose the number of friends at house u_i is v_i for $i \in [k]$. The Fundamental Theorem of Arithmetic tells us that for $v \in \{v_i\}_{i \in [k]}$, there exists a unique "prime decomposition" $v = p_1^{n_1} \cdot \ldots \cdot p_l^{n_l}$ for $l \in \mathbb{N} \setminus \{0\}$, p_i prime, and $n_i \in \mathbb{N} \setminus \{0\}$ for $i \in [l]$.

Now notice that the minimum number of snacks Umaru needs to bring from $u \rightsquigarrow v$ is $\mathsf{lcm}(u_1, \ldots, u_k)$ where $\mathsf{lcm}(a, b)$ denotes the least common multiple of a and b. Recall that if $a = p_1^{n_1} p_2^{n_2}$ and $b = p_1^{m_1} p_2^{m_2}$, then $\mathsf{lcm}(a, b) = p_1^{\mathsf{max}(n_1, m_1)} p_2^{\mathsf{max}(n_2, m_2)}$ (which obviously generalizes for any number of prime factors and any number of inputs to the lcm function).

In essence, to know the number of occurrences of each prime factor along a $u \rightsquigarrow v$ path is to know $\mathsf{lcm}(v_1, \ldots, v_k)$, so one needs only to maintain a frequency mapping of prime factors.

How many such distinct prime factors can occur? Well, notice that there can be at most 1000 type 2 snack runs and $N \leq 1000$, so this is the same as asking for a list of 2000 elements, all smaller or equal to 10^7, how many distinct prime factors can their product have? Convince yourself this is an equivalent problem before proceeding.

Some rough math suggests that this number is (very generously) bounded by, say, 7500, which will be small enough for our needs.

Factoring with Primes This is the first step of the solution. We use (for example) the Sieve of Eratosthenes to calculate whether or not each number up to 10^7 is prime, and also marking down the *smallest prime factor* of each number along the way. This is quite efficient (nearly linear).

An obvious factorization algorithm then is to repeatedly divide a number by its smallest prime factor until it is equal to 1. This lets us find the frequency of prime factors quickly. The key observation is that we can solve queries independently for each prime since the number of distinct primes will not be very large.

Path Queries We now know how to calculate the answer for a single type 1 query, but our current idea relies on essentially looping over every node in the path and fails to take into account the headache of type 2 snack runs.

The first (albeit) pessimistic observation is that `max` is not an invertible operation and we have updates, so an approach like binary lifting and lowest common ancestor (LCA) doesn't have much hope. We need something a bit heavier to handle such queries (and in particular the updates). For this we use heavy light decomposition (HLD) and a segment tree. Essentially for each distinct prime that appears as a factor of a number in the input, we use a segment tree to help us answer path queries (and do so by leveraging the power of HLD). Now updates (type 2 snack runs) are as simple as a segment tree point update.

Implementation My C++ implementation can be found here. Feel free to reach out to me at `zmasterplus#9388` (on Discord) with further questions.

I.1 Alternative Solution

There is another solution that relies on an important principle of prime factorization: the number of prime factors of any number X is at most $\log_2 X$. This lends itself to the idea that instead of keeping a count of every prime (which will have a lot of zeros), we can just store a list of each node's prime factors.

Then the queries are handled as follows:

1. Type 1 queries are handled by finding the least common ancestor of the two nodes u and v, which tells us the path from u to v. Then we can compute the LCM through the procedure described above.
2. Type 2 queries are handled by updating the list of prime factors for the desired node.

The difficult part is proving that this approach has an acceptable runtime. Here, $a_{max} = 10^7$. For each query:

1. For Type 1 queries, in the worst case, we might have to iterate through every node in the tree. Each node has at most $O(\log a_{max})$ prime factors without any Type 2 queries. Each Type 2 query adds an additional $O(\log a_{max})$ prime factors total. So the total number of prime factors, across all nodes, is $O((N+Q)\log a_{max})$.
2. Type 2 queries are easy – we just prime factorize the desired number and add them to the list of prime factors, running in $O(\log a_{max})$, where $a_{max} = 10^7$.

Therefore, the total time complexity (ignoring pre-processing, which is comparatively fast) is $O(Q \cdot (N+Q) \cdot \log a_{max})$.

J 3 Reasons to Eat Potato Chips

Author: Michelle

Setup We define the 3-tuple (a_1, a_2, a_3) where each a_i denotes the number of chips in pile i to represent a position in the game. We say a state is a "losing position" if it is not possible for the first person who starts in that position to win, and a "winning position" otherwise. Finally, we define the set of moves to be $V = \{(0,0,1),(0,1,0),(1,0,0),(1,1,1)\}$; we call $v \in V$ a *move vector*.

Insight We claim that set of all losing positions is the set of all tuples whose xor is 0. To see why this works, we really want to show two things: that a winning position always draws from some losing position, and a losing position draws from all prior winning positions. The crucial insight is that since tv is a valid move, $x - tv$ is always a winning position for some losing state x, and there always exists some losing state

$x - tv$ for a winning state x. We then convert the elements of $x - tv$ to binary and do a bit of symbol bashing to show this is true.

Tl;dr We can generate all tuples whose xor is 0 in $O(n^3)$ time, which correspond to the losing positions of the game where the first player is not able to win. Hence we will return "No" if a query is a losing position, and "Yes" otherwise. So it turns out Light will win most of the time!

K Anime Trading

Author: Arthur (bridgeminion)

L My Hero Photographia

Author: Bennett
This is mainly an implementation problem. I implemented each transformation as a function to avoid extremely messy code. Each of these functions essentially computes the resulting image in a new matrix, then replaces the global matrix with the newly calculated matrix. Some tricks:

1. You can find the rotation transformations by manually evaluating how the four corners would move.
2. After rotating, swap the n and m values to avoid future dimension issues
3. There are some issues with taking the mod of negative numbers. To avoid this, you can compute any number n

M Who is a Titan?

Author: Neo

N Portal Investigation

Author: speedycatfish
The first part of this question involves finding how many edges (portals) can be visited twice. This just boils down to finding all the strongly connected components in the graph and counting all the total number of edges in all the SCCs. This also leads to the important insight that only one clone can visit each SCC. If two clones visit an SCC, there must be an edge between a vertex (city) belonging to the first clone and the vertex belonging to the second clone that neither clone will be able to use.

The second part of the question involves finding a minimum vertex cover over the SCC graph. This is a classic problem bipartite matchign problem, where you split each vertex into v_{out} and v_{in}. Then, you connect the source to all v_{out} nodes, connect all v_{out} nodes to the v_{in} nodes they connect to in the graph, and all the v_{in} nodes to the sink. However, there's an issue. You don't need to cover SCCs with only one city because they have no edges that clones need to investigate. To remedy this issue, only connect the source and sinks to SCCs with more than once city. For SCCs with only one city, instead of connecting v_{in} to the sink, connect each v_{in} to the v_{out} of the same SCC with a capacity one edge. This allows a flow solution to "flow" through SCCS with one city that are located between SCCs that have more than one city while not specifically covering them. The final answer is the number of non-size one SCCs subtracted by max flow.