A Weather Forecast

Author: Jimmy

The solution to this problem is fairly straightforward. We can keep a counter of rainy days, and whenever we encounter a decimal value greater than or equal to 0.8, we increment this count. We can then print the count at the end.

We call decimal values **floats** or **doubles** in computer science (depending on the decimal value's precision). Most programming languages have easy ways to read in doubles - `nextDouble()` for `Scanner` in Java, `float(input())` in Python, or `cin >> x` in C++ (which will automatically read in a double if `x` is of double/float type).

B Rain Collector

Author: Prayaag

C Brownie Baking

Author: Zeki

Notice that if we want to bake a brownie of size b and we have two baking tins of size s_1, s_2 such that $s_1 \geq s_2 \geq b$, then it is always optimal to assign s_2 to bake b by a relatively simple swapping argument. What remains is figure out which brownies are worth attempting to bake. Intuitively speaking, we should always try to bake smaller brownies first (since they have the highest number of tins which they can be baked in, and of course by our argument above, the "best" tin to bake them in is the smallest). It turns out that this reasoning is sound, and these two points together give us a greedy algorithm.

So, what we can do is the following: Consider the smallest brownie, find the smallest tin which can bake that brownie. If there is no such tin, stop (since we cannot possibly expect to bake a larger brownie). Otherwise, assign that tin to the smallest brownie, increment our answer by one, and then look at the next brownie (and, of course, take the tin we used out of further consideration). This obviously suggests that we sort the brownies given to us by size. The only tricky remaining detail is finding the smallest tin which can bake each brownie. It turns out that there are multiple approaches, but perhaps the simplest approach is to use C++'s builtin `multiset` and `lower_bound` to store the sizes of the tins and repeatedly extract the best tin. My C++ implementation can be found here.

D Grumble Gym

Author: Arthur

E After School

Author: Michelle

We can sum over the solutions in each row to find our answer. Note that k occurs x times in row i, where x is the number of solutions to $\lfloor \frac{j}{i} \rfloor = k$ for $1 \leq j \leq n$. There are at most i solutions, since solving the inequality $k \leq \frac{j}{i} < k+1$ gives $j \in [ki,\ ki+i-1]$ solutions. Since j is bounded above by n, we need to take the minimum of this with $n - ki + 1$. (If this is a negative value, it means there are no solutions, so just return 0). Finally, sum over the solutions in each row to get our answer.

F Prime Precipitation

Author: Bennett
Our first goal will be to determine for each number up to H, how many prime divisors there are. We can begin this with the Sieve of Eratosthenes, in which we loop through all values $[2, \sqrt{H}]$. If the value is prime, we mark all its multiples as non-prime (this is a cool algorithm to search for if you haven't heard of it). This works because every non-prime number under H will have a prime divisor $\leq \sqrt{H}$.

We modify this a little by storing not just whether a number is prime or not but storing one prime divisor for every composite number. Now, if we know 30 is divisible by 2, we know that it has one more prime divisor than $\frac{30}{2} = 15$. Using this information, we can find how many prime divisors each number has by building our information up from 2 to H.

Once we know this, we use a similar dynamic programming approach to determine the amount of time it will take for each raindrop to fall. If a raindrop at 15 falls to 13 in one second, we know that a raindrop at 15 will take one more second than a raindrop at 13. We can similarly build up this information starting from the case $\mathbf{time}[\mathbf{1}] = \mathbf{0}$. Finally, we sum up all these values to get our answer.

G Rose and Collection

Author: Neo Wang
The optimal strategy is to keep picking roses with the smallest energy until you reach energy level E. How much energy will a rose cost? We can see that if the velocity of the monster $v < \pi$ then Rose would travel the r radius in $r/1 = r$ time. The monster would travel the radius in $\pi * r/v$ time, which means that Rose would be able to escape from the monster. Now, what if the monster can run to the other side faster than Rose? This is where the second part of the problem statement matters: if Rose chooses to run in a circle of radius $\frac{r}{v} - \epsilon, \epsilon > 0$ then Rose will be able to lap the monster, and then once they are on opposite sides, Rose would be able to run directly away from the monster.

Strongly-Connected-Components We are allowed to re-visit nodes. Observe that if we take take a subset of nodes from the graph, that forms a cycle. We can start at any node in that cycle and visit all other nodes in the cycle, ending at the starting node. If that node has multiple outgoing edges, we can first traverse all the nodes in its cycle, then choose one of the other outgoing edges to traverse to. This may lead to other cycles, and the process repeats.

We can represent each cycle as a standalone *component*. The size of a particular component is the number of nodes in the cycle. Components may point to other components, but may never form cycles (if there was a cycle between components, they could just be merged into a larger component). Finding the component is tricky, but there are standard algorithms that are intuitive and can do this easily. One such algorithm (used in the solution file) is Kosuraju's Algorithm. Fundamentally, the algorithm consists of two Depth-First Searches so the time complexity is linear. For more details about Kosuraju's Algorithm, check out this link.

DFS Once we find the strongly-connected components, we end up with a cycle-less graph consisting of components pointing at each other. This kind of graph is known as a Directed-Acyclic Graph (DAG). Since we know the number of nodes in each component, we can find the maximum number of nodes that can be reached from each component by performing DFS on each of the component's children. We store the maximum number number of nodes along any child's path in the component so if we revisit the component later, we don't have to re-perform the DFS (this keeps the time complexity linear). Finally, we just return the maximum path found from any component as our answer.

H Not-so Beautiful Painting

Author: Dylan
The main difficulty of this problem comes from the fact that the grid is very large (10^9 x 10^9). Luckily, there are quite a few ways to solve this:

Solution 1 (Sweep Line) Let's move from left to right, keeping track of the amount of paint in the current column as we go. If the current column contains the left end of a rectangle, we can add the height of the rectangle to the current amount of paint. If the previous column was the right end of a rectangle, we subtract the height of that rectangle since the current column is the first to not contain that rectangle.

To implement this efficiently, lets use a sorted map (i.e. a TreeMap in Java), which maps columns to the amount by which they change the amount of paint when moving from left to right. Anything key not in the map represents a column where no change is being made. For each rectangle x_1, y_1, x_2, y_2 the height of the rectangle h, is equal to $y_2 - y_1 + 1$. For the left side of the rectangle, we will increment the value in the map at key x_1 by h. Similarly, we will decrease the value at key $x_2 + 1$ by h. Now we can iterate over the keys in increasing order. However, we still need a way to deal with the columns with rain.

Let us also sort the list of columns with water. We can also keep a pointer at all times to the smallest index in the sorted list with a value greater than or equal to that of the current column. As we check for changes in the amount of paint in the current column, we can also increment the pointer until its condition is satisfied for the new key. The number of times it's incremented will be the number of columns in the current interval with water, k. Before processing a key in the map, if we know the previous key in the map, then we can calculate the distance between the current key and the previous key, d. Now, we want to ignore k of those columns, so the amount of paint in the current interval which isn't covered by water can be expressed as $(d - k) * h$. After summing this for all intervals, we will end up with the answer.

Solution 2 (Binary Search) The problem can also be solved with binary search. Let us first sort the M columns with rain. Now, if we iterate through all rectangles (in any order) we can binary search for the leftmost column c_{j_l} such that $c_{1_i} \leq c_{j_l}$, and binary search again for the rightmost column c_{j_r} such that $c_{j_r} \leq c_{2_i}$. Now we know that there are $c_{j_r} - c_{j_l} + 1$ columns in the rectangle which are being washed away, and $c_{2_i} - c_{1_i} - (c_{j_r} - c_{j_l})$ that are not. Thus, the area that this rectangle contributes to the solution will be $(c_{2_i} - c_{1_i} - (c_{j_r} - c_{j_l})) * (r_{2_i} - r_{1_i} - 1)$. Summing these values for all rectangles will give us the answer.

Extension How could we solve the problem if the rectangles could overlap? What about if rain could land anywhere on the painting (not just the top)? Both?

I A Rainy Delivery

Author: Yu
We can begin by representing the houses as nodes and the roads as directed edges in a graph. Note, the graph is not necessarily connected. Thus, the problem boils down to the question: "What is the maximum number of nodes that can be visited starting at any node in the graph".

J Dangerous Driving

Author: Colin

Binary Lifting Note that if two people collide, they will always move together for the rest of time. As people move forward, they will eventually end up in some cycle. Once everyone ends up in a cycle, two people at different positions can never meet since they will move at the same rate around the cycle. Therefore, if we

compute their positions after a large number of moves after everyone has settled into their cycles, everyone will be grouped together with everyone they have met.

It takes at most n steps for everyone to end up in a cycle, but we can compute everyone's positions after any large number of days that is at least n and still get a correct answer. Let $f(x, 2^i)$ be the position of person x after 2^i days. To calculate $f(x, 2^{i+1})$, we let $y = f(x, 2^i)$ represent half of the journey. For the rest, we can calculate where y would end up after 2^i days, which is $f(y, 2^i)$. Overall, this means that $f(x, 2^{i+1}) = f(f(x, 2^i), 2^i)$. Computing this for $i = 1$ to $i = 30$ gives us the positions for everyone after 2^{30} steps, which is larger than n.

Finally, if there are c people at node x after 2^{30}, then they form $c(n-c)$ pairs with people outside of the node. We can sum this up and then divide by 2 to get the total number of pairs.

Sample solution

K Rain on Birthday

Author: Aaryan

Let $x \otimes y$ be the XOR of x and y. First of all, we note that if we can construct $c_1 = \bigoplus_{c \in S_1} c$ and $c_2 = \bigoplus_{c \in S_2} c$, then $c_1 \oplus c_2 = \bigoplus_{c \in S'} c$ where $S' = (S_1 \cup S_2) - (S_1 \cap S_2)$. In other words, if we can construct both c_1 and c_2 by picking certain subsets of chemicals, we can always construct $c_1 \oplus c_2$ even if the sets overlap.

Since we know that all the values for c are in the interval $[0, 2^30)$, each number is equivalent to some vector in $\mathbb{Z}_2^{30}$, with XOR corresponding to addition in this vector space. The set of possible numbers that we can create is some closed subspace of this vector space. To keep track of this subspace, we can maintain a basis of the subspace.

In general, suppose we have some arbitrary set of independent vectors. To check if a vector v is in the span of the set, we can first run row reduction on the grid so that each vector has a unique pivot. Using these pivots, we can greedily pick the vectors required to construct each bit of v. We can use this procedure to add v into the basis if we are unable to construct it from the current basis.

To answer the queries of type 2, we note that a tells us which bits of the number we construct must match the corresponding bits in b. To do this, we can again run row reduction to compute pivots, but preferring pivots for the required bits over the other bits that don't matter. We can then iterate over the pivots for the required bits that must be set and greedily pick the vector if needed.

Overall, row reduction is $O(30^3)$, so the time complexity is $O(30^3 q)$. In practice, this runs fairly fast, with a C++ implementation taking about 260ms, and a PyPy implementation running within 1300ms. A sample C++ implementation can be found here.

For more information about the general approach, read the XOR Basis blog, linked here.