2nd Universal Cup Stage 6: Warsaw

editorial

A. Digital Nim

Let's call the number n, which is the number of stones in the pile, a "state". For some states the person who makes the first move will win, while for others, the person who makes the second move will win. We'll call the former ones "winning states" and the latter "losing states". Let's also denote by $ds(x)$ the sum of digits in the decimal representation of x.

State $n = 0$ is losing and every other state is losing if and only if any of the states $[n - ds(n), n)$ is losing. Thus, if we move from smaller to bigger pile sizes, we are interested only in the last losing state. Let's denote by $last(x)$ the minimum possible value of $x - y$, such that $y < x$ and y is a losing state. If $last(x) \leq ds(x)$, then $last(x + 1) = 1$, otherwise $last(x + 1) = last(x) + 1$. To solve a testcase we need to check if $last(n + 1) = 1$.

Let's assume that we know the value of $last(\ell \cdot 10^k)$ and we want to know the value of $last((\ell+1) \cdot 10^k)$. If $k = 0$, we can do it as quickly as we can calculate $ds(\ell)$. If $k > 0$, we can split such calculation into 10 smaller calculations: $last(10\ell \cdot 10^{k-1}) \to last((10\ell+1) \cdot 10^{k-1}) \to last((10\ell+2) \cdot 10^{k-1}) \to \ldots \to last((10\ell+9) \cdot 10^{k-1}) \to last((10\ell + 10) \cdot 10^{k-1})$.

Now we can notice, that the exact value of ℓ doesn't matter, only the values of $ds(\ell)$ and $last(\ell \cdot 10^k)$ are important, so we can use memoization. There are $\mathcal{O}(\log(n))$ possible values of k, $\mathcal{O}(\log(n))$ (times 10) values of $ds(\ell)$ and the same number of possible values of $last()$. We also split each calculation into 10 smaller ones, so the whole preprocessing takes time $\mathcal{O}((base \cdot \log_{base}(n))^3)$. Using it we can solve each testcase in time $\mathcal{O}(base \cdot \log_{base}(n))$.

B. The Doubling Game 2

Firstly we have to notice, that if $max(v)$ denotes the highest number of tokens that can be gathered in vertex v, then the sum of values $max()$ for all vertices is $\mathcal{O}(n \log(n))$ (the proof is not that hard, focus on the tree compressed to vertices with $max() \geq x$ and on vertices with degree ≤ 2 in it). Instead of the final arrangement of tokens, we will focus on deciding, for each edge, how many tokens (if any) will travel through it and in which direction (for each correct arragement of tokens there is an unique way to obtain it).

The condition for the arragement of directed weights on edges to be correct, is that for each vertex either there are only incoming weights $1, 2, 4, 8, 16, 32 \ldots$ (any length is ok, but only prefixes of powers of 2 are good) or there is one extra outgoing edge with weight equal to the sum of incoming weights plus one.

Let's do tree DP. For each vertex v let's sort its children in nondecreasing order of $max()$. While scaning the children one by one, we are interested in the exact mask of weights that are already going into v and an additional information if we've already decided the weight that will be outgoing of v. The trick is to notice, that if the last scanned child is u, all the information that we need to store is of size $\mathcal{O}(max(u))$. We need one extra logarithm for transitions, because for each child we have to decide the weight of the edge to/from it. The whole complexity will be $\mathcal{O}(n \cdot \log^2(n))$.

Good trick to help with implementation is to avoid calculating the exact values of $max()$, but notice that for each vertex you are returning something like two lists, so simply cut the zeroes from their ends and treat their lengths as parameters to sort the children.

C. Cliques

Notice that if in a collection of paths each pair has a non-empty intersetion, then the intersection of all paths is also non-empty. Let's fix vertex v as LCA of such intersection for some clique (use any root). How many subsets of paths have LCA exactly here? For sure each of the paths has to contain v. Let's say there are x such paths. But it's possible, that if we pick some subset, all the paths will also contain at least one vertex above v. So let's say that there are y paths that contain v and its parent. Then there are $2^x - 2^y$ sets of paths for which v is LCA of their intersection. So for each vertex we need to store two numbers, 2^x and 2^y. Adding and removing paths is easy, cause we need to multiply/divide by 2 everything on $\mathcal{O}(1)$ paths in the tree. Using heavy-light decomposition, we can solve the problem in time $\mathcal{O}(n \log^2(n))$.

D. Colonization

Let's learn how to calculate the number of colonizers for a given tree. Let's assume that they will start in the given vertex and make that vertex the root (finally we will assume that for each vertex and the minimum of the answers). Then one of the colonizers will wait in the root and all other colonizers will colonize each of the subtrees recursively. The only exception is the last subtree of the root, because the colonizer that was waiting in the root can go with the rest of colonizers and help them colonize this last subtree.

Above problem can be solved with a simple tree DP:

- $DP(leaf) = 1$
- $DP(v) = \begin{cases} \max_c dp(c) & \text{when the max value appears only once} \\ \max_c dp(c) + 1 & \text{in other cases} \end{cases}$

Let's denote by D_v the $DP(v)$ calculated assuming that v is the root. We need two observations:

- $max_{a,b \in \mathcal{T}} |D_a - D_b| \leq 1$
- $max_{v \in \mathcal{T}} D_v = \mathcal{O}(\log(n))$

Let's now revisit how to calculate unlabeled unrooted trees. Each such tree has only one centroid (ok, it can have two). By $trees(m)$ let's denote the number of unlabeled rooted trees with m vertices, such that each subtree of the root has the size smaller than $\frac{n}{2}$. If n is odd, the answer is $trees(n)$. If n is even, the answer is $trees(n) + \binom{trees(\frac{n}{2})+1}{2}$. All values of $trees(m)$ for $m \leq n$ can be calculated in time $\mathcal{O}(n^2 \log(n))$.

Above DP can be modified to calculate the number of trees with fixed D_{root} and the number of children of the root with maximum value of D_c (we only need to know if there are 1, 2 or more such children). Let's denote it by $trees(m, d, cnt)$, where $D_{root} = d$ and $cnt = 3$ means 3 or more children. We can calculate it in time $\mathcal{O}(n^2 \log^3(n))$ or even $\mathcal{O}(n^2 \log^2(n))$.

The above DP includes also trees for which the number of colonizers could be lower if the move the root. Let's similarly denote by $bad(m, d, x)$ the number of rooted unlabeled trees, for which $D_{root} = d+1$, but the overall result will be equal to d if we attach to the root a tree with (rooted) answer less than or equal to x, but it won't be equal to d if we attach a tree with (rooted) answer greater than x.

This DP can also (using the values of trees()) be calculated in time $\mathcal{O}(n^2 \log^2(n))$ and using it we can get the answer for each k from $[1, n]$.

E. Bus Lines

Focus on calculating the sum of distances from some vertex v. Denote by R_i the set of vertices reachable from v with at most i buses (R_0 contains only v). The number of vertices with the distance greater than x is equal to $n - |R_x|$ and if we look at sum of $n - |R_x|$ for all x, we will get the sum of distances to all the vertices (each vertex is counted as many times as the distance to it).

Now look at some components R_i and R_{i+1} and on edges goining out of them (think that they are directed outwards). Each edge going out of R_i **independently of other such edges** transforms into some

(possibly zero) edges going out of R_{i+1}. To find these edges for some edge $u \to w$, we need to look at all the paths crossing $u \to w$, find the subtree (on the correct side of the edge) that they span and take all the edges going out of it.

We wanted to calculate $n - |R_i|$, but we can also look at it as the sum of sizes of subtrees that the outgoing edges point to. So we want a DP for directed edges which will be equal to the number of vertices on this side of the edge plus the sum of DPs for all the edges it will transform to. Final answer for a vertex v will be equal to sum of DP values for all edges going out of it.

Now let's root the tree and think of separately of edges going upwards (towards the root) and downwards. Calculating the sizes of subtrees is easy, nextly we need to do three phases:

- Add all downward DPs to appropriate downward DPs. It has to be done from bottom to top.
- Add all downward DPs to appropriate upward DPs.
- Add all upward DPs to appropriate upward DPs. It has to be done from top to bottom.

The first phase can be done with some tricky amortization or in a bit harder way which is very similar to the second phase.

The second can be done in any order, so let's do it from bottom to top. For each edge we want to maintain the subtree above it spanned by all the paths that cross this edge. This can be done with smaller to larger sets and some standard tree stuff. We also need to maintain appropriate sums for such trees, but we can use the fact that if we store the nodes sorted by preorder and consider the paths between all neighboring (in preorder) pairs of vertices (also counting first and last as a pair) we will store information about each edge exactly twice.

The last phase is the easiest one, as to each upward DP we need to add at most one other upward DP.

This gives us final complexity $\mathcal{O}((n+m)\log^2(n+m))$.

F. Max-Min

We will deal with calculatin sum of maximums and minimums separately, and if we multiply the array by -1 we can deal with both of them with the same code. So let's focus on calculating the sum of maximums.

Calculating it for a static array is a standard problem, for example if we have a segment tree, we can find maximum, notice that all the intervals that contain it have the same maximum, and split into two smaller intervals.

Now let's assume that an k-th element increases. On which intervals will the maximum change? Let's denote by $left(k)$ the closest position to the left with a value strictly greater than a_k (before the change) (if there are no such positions, set $left(k) = 0$) and similarly let's define $right(k)$. The maximum will increase by 1 on exactly $(k - left(k)) \cdot (right(k) - k)$ intervals, so we can keep track of the sum of maximums on all intervals and update it. Decreasing a value is very similar.

So we need to be able to calculate values of $left()$ and $right()$ quickly. Luckily it's easy, we only need a segment tree which stores maximums in each segment. The final complexity if $\mathcal{O}(n\log(n))$.

G. Matrix Inverse

Let's denote by k the maximum number of cells that are allowed to be incorrect in B.

Multiplying matrices is slow, in CP conditions it requires $\mathcal{O}(n^3)$ time. Do you know what is fast? Multiplying a vector by a matrix. Let's pick a random vector, multiply it by A and nextly multiply it by B (if you feel unsafe, you can do it a few times). Let's also pick random vector and multiply if from the other side by A and B. This way (in time $\mathcal{O}(n^2)$)we can find the sets of at most k rows and columns and we know that the k cells that we are looking for are among k^2 cells in the intersection of these rows and columns.

Let's focus on one row in B. We know what the result of multiplying this row by A should be. We also know the set of k positions in it that we possibly have to change. We can set a variable for each of the positions that we possibly will change, and we have a system with k variables and n linear equations (calculating this system requires $\mathcal{O}(n^2)$ time). We can solve it in $\mathcal{O}(nk^2)$ time.

Doing so for each of k rows requires $\mathcal{O}(n^2k + nk^3)$ time (and could be reduced further, which wasn't required).

H. Inverse Problem

Firstly we need to notice that (if n is fixed) the answer depends only on the multiset of degrees of vertices in the tree. To be precise, there exist a function $f(n)$ and function $g(n, d)$ (such that $g(n, 0) = 1$) such that result is equal to $f(n) \cdot \prod_{v \in V} g(n, deg(v) - 1)$. Nextly we have to notice that because of $g(n, 0) = 1$ and because the sum of $deg(v) - 1$ is equal to $n - 2$, we need to split $n - 2$ into sum of any number of numbers so that the above product is equal to what we want.

Why do we do that? We hope that for each partition that we consider the above product will be more or less random, so it will be likely that all the remainders will be covered quickly, which turns out to be true. There are $M - 1$ (M is equal to $10^9 + 7$) remainders and we need to try about $\mathcal{O}(M \log(M))$ partitions to cover all of them, so it's too slow to try each partition.

Instead of that, let's iterate over n. If n is fixed let's split all the numbers from 1 to $n - 2$ into two sets, for example let $A = \{1, 2, 3, 4, 5, 6\}$ and $B = \{7, 8, 9, \ldots, n - 3, n - 2\}$. Now for both of them generate every possible partition which sums up to at most $n - 2$. We've chosen such a way to split, so that the amount of partitions for both sets is more or less equal. Together with such partitions, let's calculate appropriate products (and for convenience the products for the first set can be already multiplied by $f(n)$ and for the second set, instead of the product, we we'll calculate its modular inverse).

Let's say that we want the sizes of partitions from sets A and B to be equal to S_A and S_B respectively (with $S_A + S_B = n - 2$) and the products to be equal to P_A and P_B respectively (with $P_A \cdot P_B \equiv R \pmod{M}$). Let's iterate over S_B and over the exact partition of numbers from B. Now we want exactly what partition from A we are looking for. So before we iterate, we can build a data structure which for given S_A will tell us if there is a partition with appropriate product. To do this it's good enough to use a huge bitset.

Knowing the multiset of degrees it's easy to construct any tree.

Such meet in the middle lets us solve the problem even for the greatest possible n, which turns out to be equal to 125.

I. Boxes

Sort the boxes to make the array nondecreasing. Iterate over i from 1 to n. Try to put the i-th box into the leftmost box that still has place for it. Done.

J. Sequence Folding

Consider folding the sequence. If two zeroes or two ones collide, don't do anything. If a zero collides with one, increase the result by one and replace them with a... question mark. If two question marks collide, leave them as question marks. If a question mark collides with digit, change it to this digit (and don't pay, you already did that). Continue folding untill the length of the sequence becomes one.

K. Jump Graph

Add the elements in the order from 1 to n. Each number will merge two intervals of smaller values. Imagine adding a number v will merge the intervals L and R and that directly to the left of L there is a number ℓ (which is greater than v) and directly to the left of R there is a number r (which is also greater than v). It's possible that ℓ or r doesn't exists, we'll deal with that later.

So in the sequence we have something like $\ell LvRr$. Now notice, that to get from anything in L to anything in R, your first jump will be to either v or r (the later might not be possible). Also notice that you can jump from v to r and from r to v. This mean that the distances from v and from r to anything in R must differ at most by 1. So let's store the sum of minimums of distances from v or r to everything in R and the

number of elements in R that are closer from v/closer from r/in the same distance from v and r. Now for everything in L we need to increase its final result by some known value. We can do it in constant time and recover all the results in linear time at the end. The same goes for going from R to L and it's similar for going from v to L or R.

Using the values that we calculated for L and R we can calculate everything that we need for LvR. To deal with nonexisting elements at the beginning and the end of the sequence you can modify the returned values, so that they always point to the existing element as the one closer to them. If we use lists, we can finish with linear complexity, but DSU also works fine.

L. Spectacle

Sort the players by rating. Now it's easy to prove that we will pair only neighboring players. So let's sort the differences between neighboring players and "activate" these pairs one by one. To activate a pair we will use DSU and we will merge these players into one component. For a fixed set of allowed pairs, what's the largest number of pairs we can create? If there is a component of size s, we can create at most $\lfloor \frac{s}{2} \rfloor$ pairs in it. So we can maintain this sum over all components. With it, for each rating difference, we can know the maximum number of pairs for this difference. From this we can for each number of pairs know the minimum possible difference, which solves the problem.

M. Pattern Search

Let's denote by t_x the number of occurrences of the character x in t, and in the same way let's define s_x. Of course, if $s_x < t_x$ for any x, the answer is 0. Let's iterate over the number of times a period of t will repeat at its beginning and denote it by k. Also let's assume that each character x appears p_x times in the period and q_x times at the end of t (in the possibly unfinished occurrence of period). This gives us equations that must hold for each i:

$$t_i = p_i \cdot k + q_i$$

$$p_i \geq q_i$$

With fixed p and q the answer is equal to:

$$\min_i \left(1 + \left\lfloor \frac{s_i - t_i}{p_i} \right\rfloor \right)$$

So we want to minimize p_i for each i independently and we can do it in constant time. Final complexity can be equal for example to $\mathcal{O}((|s|+|t|) \cdot |A|)$, where $|A|= 26$.