

A. All-Star

Author: Pavle Martinović
Solved by: 77/82
First to solve: *UniBuc_KoalifiedKoalas*

Let d be the largest degree of a node in the tree. We will prove that the answer is $n - 1 - d$.

First, we can immediately notice that if we make a move on outhouses x, y and z , we will increase the degree of x by 1 and decrease the degree of y by 1. This means that we can increase the maximum degree in the tree by at most 1 in each move. Since a star has maximum degree $n - 1$, we need to make at least $n - 1 - d$ moves.

Now we actually need to find a sequence of $n - 1 - d$ moves. Let s be a vertex with degree d . Suppose that the tree isn't a star with center s . Then there exists a path of length 2 starting at s ; say $s - x - y$. Performing a move on these vertices disconnects x and y , and connects y to s , increasing the degree of s by 1. So, as long as the tree isn't a star, we can increase the degree of s by 1, and so we can perform $n - 1 - d$ moves to get s to have degree $n - 1$.

We can either simulate the above algorithm, taking $O(N)$ to find the move in each step (leading to total time complexity $O(N^2)$). Alternatively, one can easily notice that running a DFS algorithm from s and outputting $s - x - y$ for every edge we encounter xy not incident to s in order will essentially do the same thing as the above paragraph describes in $O(N)$ time (this can be proven by induction, say).

B. NonZero PrefSuf Sums

Author: Anton Trygub
Solved by: 1/1
First to solve: *Fizičari*

Let's analyze when does array $[a_1, a_2, \dots, a_n]$ satisfy the conditions of the problem.

1. If $a_1 + a_2 + \dots + a_n = 0$, then it's a NO. Now, assume $a_1 + a_2 + \dots + a_n > 0$ (otherwise just negate all elements).
2. If array contains at most one nonzero value, it's a NO.
3. Let the sum of positive elements be P , and the sum of negative elements be $-N$, with $P > N$. Denote $S = P - N$. If we order all positive elements first, and then all negative, then all prefix sums will be positive, so we only have to worry about suffix sums. Alternately, we only need to order positive elements so that no prefix sum equals exactly S .
4. If not all positive elements are equal, then it's always possible to order positive elements in such a way, so it's a YES. Indeed: assume there are some positive elements $x \neq y$. First, as many other positive elements as you can so that no prefix sum is S . If we could order all of them, then we put x, y in one order or another, making sure new prefix sum is also not S . Otherwise, there are some elements remaining, but we can't append any more without getting sum exactly S , which means all remaining elements are equal to some t . Then put x or y (whichever is $\neq t$), then t , then all remaining positive elements: we successfully avoided S .
5. Now assume all positive elements are equal to some x . If S is not divisible by x , we can still put all positive elements, and then all others, so in that case the answer is also YES. So now we assume that P, S, N are divisible by x .
6. Assume some negative number y isn't divisible by x . Then let's put y first, then all xs , then all remaining elements. Note that every prefix sum is either positive or not divisible by x , and every suffix sum is also either negative or not divisible by x , so the answer in this case is also YES.
7. Otherwise, all numbers are divisible by x . So let's divide everything by x , and assume that $x = 1$ from now on.
8. Let's think about the prefix sums a bit more deeply. We want each prefix sum (except the first and last one) to be $\neq 0, S$. They also start with 0, and end with S . In addition, whenever prefix sums increase, they increase by exactly 1 (since that's the only positive elements we have). So, if they ever get negative, in order to get to S , they would have to cross 0 at some point.
9. So, the arranging process looks as follows. We start with 0, and then at every step we can either use 1, increasing sum by 1, or use some negative number $-x$, decreasing sum by x . We cannot ever get to 0 or S (except first or last position), so we have to always remain in the $[1, S - 1]$ range. So, if there exists some negative number $< -(S - 2)$, the answer has to be NO.
10. It's easy to see that otherwise such an arrangement always exists: just place 1s until sum doesn't exceed $S - 1$, and then use some negative number.

Now, we need to write *dp* for this. Once again, let's break this down into small steps.

1. Total number of arrays is $(2m + 1)^n$.
2. Calculate number of arrays $\in [-m, m]^n$ with sum 0, you can do this in $O(mn^2)$, subtract it.
3. Need to calculate only number of bad arrays with positive sum (then multiply it by 2 and subtract).

4. Iterate over positive element x . We know all elements have to be divisible by x , so we need to solve the instance with positive elements equal to 1, and all elements now being in $[-\frac{m}{x}, \frac{m}{x}]$.
5. Iterate over: number of ones P , and sum of negative elements $-N$ with $N < P$. Then all negative elements have to be up to $S - 2 = P - N - 2$. So, we need to count the number of arrays of length $n - P$ of nonnegative numbers with sum N with elements not exceeding $P - N - 2$.
6. We will use more general dp for this: $dp[len][sum][lim]$: the number of arrays of length len with sum sum and all elements in $[0, lim]$. We can code this in $O(MAXX^4)$, where $MAXX = \max(n, m)$. Indeed, transitions are simple: calculate values for lim from 0 to $MAXX$, every time iterate how many values equal to lim you are adding.

Total runtime: $\max(m, n)^4$ with a very good constant (can actually be done faster but we reduced constraints enough).

C. Duloc Network

Author: Adrian Miclaus
Solved by: 16/25
First to solve: [UAIC] Washbin

Let $X \subseteq V$. We denote by $f(X)$ the number returned by the interactor. Let $A, B \subseteq V$ be two sets of vertices such that $A \cap B = \emptyset$. If $f(A) + f(B) \neq f(A \cup B)$, there are two cases:

- $\exists x \in A$ and $y \in B$ such that $(x, y) \in E$.
- $\exists z \in V$ such that $z \notin (A \cup B)$ and $\exists x \in A$ and $\exists y \in B$ such that $(x, z), (y, z) \in E$.

Both cases represent the fact that x and y are connected. In fact, that also means that $dist(x, y) \leq 2$ where $dist$ is the distance between two vertices in the original graph.

Thus, we can start with $A = \{1\}$ and $B = \{2, 3, \dots, |V|\}$ and binary search the next vertex that can be added to A such that we know that all vertices of A are in the same connected component. Firstly, we can check that $f(A) \neq 0$, the graph is not connected otherwise. At each step of the binary search we divide the set B into two sets B_1, B_2 such that B_1 has the first $\frac{|B|}{2}$ elements, and B_2 the last $|B| - \frac{|B|}{2}$. If $f(A) + f(B_1) \neq f(A \cup B_1)$, then we continue the search for the vertex to add to A in B_1 , otherwise in B_2 . Then we know that the distance from the vertex we found to some vertex in A is at most 2 and that means they are in the same connected component, so we can add that vertex to A . The number of queries used is at most $2 \cdot |V| \log |V|$.

D. Donkey and Puss in Boots

Author: Yarema Stiahar
Solved by: 90/90
First to solve: *RAF 100011*

Puss in Boots in his turn can take candies from all piles, so he can take all candies. Puss in Boots can lose only if he cannot make n moves in his turn. This would happen if the total number of candies across all piles is less than n . Therefore, the optimal strategy for Donkey is to take all the candies from the biggest pile. There is also the case where all piles start with 0 candies. In this case, neither player can make a move, and the game ends immediately, with Donkey's defeat.

E. Shrooks

Author: Anton Trygub
Solved by: 0/0
First to solve: N/A

The key idea is to try to characterize all good placements of rooks. It turns out they all have to lie on a rhombus of some sort. Note that the condition on all Manhattan distances being at most n is equivalent to the following:

- There exist some integers A, B , such that for every rook (x, y) we have

$$A \leq x - y \leq A + n$$

$$B \leq x + y \leq B + n$$

Note that it follows that $A + B \leq 2x = (x - y) + (x + y) \leq A + B + 2n$ for any rook (x, y) , so $\frac{A+B}{2} \leq x \leq \frac{A+B}{2} + n$. Since x can be any number, we get

$$\frac{A+B}{2} \leq 1, \frac{A+B}{2} + n \geq n \implies 0 \leq \frac{A+B}{2} \leq 1 \implies 0 \leq A + B \leq 2$$

Similarly, we know that $(B - A) - n \leq 2y = (x + y) - (x - y) \leq (B - A) + n \implies \frac{B-A}{2} - \frac{n}{2} \leq y \leq \frac{B-A}{2} + \frac{n}{2}$ for any rook (x, y) . Once again, it follows that

$$\frac{B-A}{2} - \frac{n}{2} \leq 1, \frac{B-A}{2} + \frac{n}{2} \geq n \implies \frac{n}{2} \leq \frac{B-A}{2} \leq \frac{n}{2} + 1 \implies n \leq B - A \leq n + 2$$

Consider the case of odd $n = 2k + 1$ first. Then there are 4 cases: $(A, B) \in \{(-k - 1, k + 1), (-k, k + 1), (-k - 1, k + 2), (-k, k + 2)\}$. Each of these cases is basically defining some rhombus inside which every rook has to be; all of them are symmetric in the sense that they can be obtained by rotating the board by 90° . So, we will count the number of configurations working for the case $(A, B) = (-k - 1, k + 1)$, and for other cases just rotate the board.

From $(A, B) = (-k - 1, k + 1)$ it follows that for rook (n, y) we have

$$-k - 1 \leq n - y \leq -k - 1 + n, k + 1 \leq n + y \leq k + 1 + n \implies k + 1 \leq y \leq k + 1 \implies y = k + 1$$

So, there definitely has to be a rook at $(n, k + 1)$: at the middle of the bottom column. It's not hard to deduce where other rooks have to be.

Now it's not hard to fill where the remaining rooks will be. Consider, for example, columns 1 and n . From current constraints, it's easy to see that corresponding rooks in them have to be in rows k or $k + 1$. So, it's either $\{(k, 1), (k + 1, n)\}$ or $\{(k + 1, 1), (k, n)\}$. Then we can look at columns 2, $n - 1$, and deduce that it's either $\{(k - 1, 2), (k + 2, n - 1)\}$ or $\{(k + 2, 2), (k - 1, n - 1)\}$, and so on: all other rooks are split into pairs, for each of which there are two placement choices. We can easily find the number of placements that satisfy already placed rooks.

However, we need to be careful to not double count any configurations. Fortunately, there are only 4 of them that satisfy two choices of A, B : one where rooks are placed at

$$\{(1, k + 1), (2, k), (3, k - 1), \dots, (k + 1, 1), (k + 2, n), (k + 3, n - 1), \dots, (n, k + 1)\}$$

And 4 its rotations. So just subtract whichever of these placements satisfy already placed rooks.

Runtime $O(n)$.

The analysis for the case of even $n = 2k$ is similar. Once again, there are 5 cases for (A, B) :

$$\{(-k, k), (-k, k + 2), (-k - 1, k + 1), (-k + 1, k + 1), (-k, k + 1)\}$$

Here the first four are also rotations as each other, and the last one is a bit different.

The base configuration for the first 4 looks as follows: there are two rooks at cells $(1, k+1), (k+1, 1)$, and for $2 \leq i \leq k$ there are two rooks at $\{(i, k+2-i), (2k+2-i, k+i-2)\}$ or at $\{(i, k+i-2), (2k+2-i, k+2-i)\}$. Once again, iterate over these 4 rotations.

$(-k, k+1)$ corresponds to the following:

For every $1 \leq i \leq k$, there are two rooks at $\{(i, k+1-i), (2k+1-i, k+i)\}$ or at $\{(i, k-i), (2k+1-i, k+1-i)\}$.

Once again, we need to avoid double counting. In this case, configurations that might be double counted are of form:

$$\{(1, k), (2, k-1), (3, k-2), \dots, (k, 1), (k+1, n), (k+2, n-1), \dots, (n, k+1)\}$$

And their rotations.

Runtime $O(n)$.

F. Magical Bags

Author: Roman Bilyi
Solved by: 9/16
First to solve: *Infinity*

The condition on is equivalent to:

- Two bags X and Y are good if and only if $\max(X) > \min(Y)$ and $\max(Y) > \min(X)$.

We can create a segment $[\min(X), \max(X)]$ that characterize a bag. Two bags are good iff their corresponding segments intersect (have at least one common point).

1. It's never optimal to leave more than 2 objects in a bag. That's because to check all conditions, we only use at most 2 values: minimum and maximum in the bag. If some bag contains 3 values $a < b < c$, we will never use b to check whether a pair of bags is good and therefore could be removed.
2. So all bags should contain either 1 or 2 object, and we want to maximize the number of bags with 1 object. For now, consider the case when all bags have at least 2 objects initially. Bags with one object don't create any corner cases, it is just easier to explain the solution.
3. If a pair of bags isn't good initially, it can't become good after removing objects. It also means that if we leave 2 objects in a bag X , it should the minimum and the maximum. We want the segment corresponding to bag X to intersect with some other segments, and there are no conditions that it can't intersect with some other segments. That means it's optimal to choose the segment as large as we can – $[\min(X), \max(X)]$.
4. Let's call a bag **promising** if we can leave 1 object in this bag and 2 objects in all other bags. Let's find whether the bag A is promising. How to find whether we can leave the value $x \in A$ as the only remaining value?
5. We can't leave x as the only remaining value if and only if there exist bag B such that $\min(A) < \max(B) < x$ or there exist bag C such that $x < \min(C) < \max(A)$. In such case, bag A and bag B or C were initially good but not good anymore. To implement this, we can sort all values $\min(X)$ for all bags X , sort all values $\max(X)$ and use binary search to find whether a value in a given range exist. Note that we should check every value $x \in A$, not only $x = \min(A)$ or $x = \max(A)$.
6. Now, let's call a bag **special** if we leave 1 object in it. Obviously, each special bag should be promising. We know that the condition holds for each pair of two non-special bags. Also, the condition holds and for each pair of special and non-special bag by the definition of promising.
7. Any pair of special bags can't be good initially. If we leave only one value $a \in A$ and only one value $b \in B$, $a < b$ and $a > b$ can't hold simultaneously.
8. This means that we should choose the largest subsequence of promising bags, such that each pair of chosen bags is not good. If we denote each bag by its segment, we should find the largest subsequence of non-intersecting segment. Such problem is well-known and can be solved greedily: sort all segments by right end and choose a segment if it doesn't intersect with the last chosen segment.
9. If we have bags that contain 1 element initially, those will be promising and then chosen, so no corner case required.

The complexity of the solution is $O(m \log n)$, where m is the total amount of objects.

G. Shrek's Song of the Swamp

Author: Adrian Miclaus
Solved by: 63/72
First to solve: LNTU_IPZ_3

In this problem, we need to determine the longest subsequence with the property that it can be divided into blocks of equal elements of length at least 2. The main observation is that such a subsequence can be divided into blocks of equal elements of length 2 or 3. Thus, we can solve the problem by dynamic programming. Let $dp[i]$ be the length of the longest subsequence with the property mentioned above in the prefix s_1, s_2, \dots, s_i . We start by initializing $dp[0] = dp[1] = 0$. Then, $dp[i]$ can be computed with the following recurrence:

$$dp[i] = \max \begin{cases} dp[i-1] \\ dp[i-x] + 1, x = \max\{j | j \in \{1, 2, \dots, i-1\} \text{ and } s[j] = s[i]\} \\ dp[i-x-1] + 2, x = \max\{j | j \in \{1, 2, \dots, i-1\} \text{ and } s[j] = s[i]\} \end{cases}$$

To fill the array dp we need to iterate from 1 to n and keep the last occurrence of each element. This can be done with an `unordered_map`/ `map` resulting in an algorithm of complexity $O(n)/O(n \log n)$. The answer is $dp[n]$.

H. Shreckless

Author: Anton Trygub
Solved by: 39/50
First to solve: RAF 100011

Consider two adjacent columns, assume in sorted order they are $a_1 \leq a_2 \leq \dots \leq a_n$ and $b_1 \leq b_2 \leq \dots \leq b_n$. For any permutation p , what's the maximum number of indices i such that $a_i > b_{p_i}$? (Therefore making the corresponding row not sorted).

Let's see how to check if we can get at least k such indices. It makes sense to pair k the largest elements of a with k smallest elements of b . It's also clear that we should be pairing them in increasing order: $(a_{n-k+1}, b_1), \dots, (a_n, b_k)$. Then we can get k pairs iff $a_{n-k+i} > b_i$ for all $1 \leq i \leq k$.

Now, we can find the largest number of such pairs we can form with binary search over k . This would work in $O(n \log n)$. We will denote this value by $f(a, b)$. Denote the columns as a_1, a_2, \dots, a_m . Clearly, we cannot make more than $f(a_1, a_2) + f(a_2, a_3), \dots, f(a_{m-1}, a_m)$ unsorted rows. So, if $f(a_1, a_2) + f(a_2, a_3), \dots, f(a_{m-1}, a_m) < n$, the answer is NO.

It turns out that if $f(a_1, a_2) + f(a_2, a_3), \dots, f(a_{m-1}, a_m) \geq n$, we can make all rows not sorted! Algorithm is very simple. Arrange elements of the 1-st column arbitrarily. Then, for $i = 2, \dots, m$, do the following:

Assume there are x sorted rows remaining (for $i = 2$ we have $x = n$). We assume x largest elements of a_{i-1} are in precisely those columns (we will see why later). Now, arrange the elements of a_i as follows:

1. Put $\min(x, f(a_{i-1}, a_i))$ smallest elements of column a_i next to $\min(x, f(a_{i-1}, a_i))$ largest elements of column a_{i-1} , as discussed before, making those rows not sorted.
2. Put $x - \min(x, f(a_{i-1}, a_i))$ largest elements of column a_i in remaining sorted rows.
3. Put other elements arbitrarily.

This way, we will get $\min(n, f(a_1, a_2) + f(a_2, a_3), \dots, f(a_{m-1}, a_m))$ unsorted rows.

Runtime: $O(nm \log n)$.

I. Donkey, Keep Watch

Author: Pavle Martinović
Solved by: 0/0
First to solve: N/A

We split into two cases, whether $|s|$ is even or odd.

Case 1: $|s|$ is even. Let's look at each bit independently. If the bit is set in any element of s , we know it will exist in the OR. This bit will exist in the AND if it's set in every element of S , and since $|s|$ is even, then it won't be set in the XOR. So every bit set in the OR is set in at most one of AND, XOR, so $\text{AND}(s) + \text{XOR}(s) \leq \text{OR}(s)$. The equality is attained only when each bit is set everywhere, nowhere or in an odd number of elements (these are three distinct cases).

Let's fix a ternary mask of size 14 (being the log of $\max a$), containing 0,1 and ?. This mask represents that we want to select some subsequence such that:

- if $\text{mask}[i] = 1$, then all elements of this subsequence have bit i set;
- if $\text{mask}[i] = 0$, then no elements of this subsequence have bit i set;
- if $\text{mask}[i] = ?$, then an odd number of elements of this subsequence have bit i set.

We also have to ensure that the set is of even size. Let B be the multiset of elements of a that follow the pattern made by the ternary mask. Now we just need to find the number of even subsets of B that such that their xor is equal to one on the ? positions. If our mask has m positions with a ?, suppose we look at every element in B as a vector in \mathbb{F}_2^{m+1} with entries corresponding to entries of those elements in the ? positions, and also one 1 at the end to keep track of parity. We need to find the number of subsets of these vectors summing to $(1, 1, \dots, 1, 1, 0)$ (the ones at the first positions mean that the XOR is 1 on the ? positions; the zero at the end means that there is an even number of elements). By linear algebra (for example, the Rank-Nullity Theorem) we know that the number of such subsets is either 0 or $2^{|B|-d}$, where d is the dimension of the subspace generated by the elements of B . The way we can check whether the answer is 0 we need to find the basis generated by set B (by Gaussian elimination), and simply check whether we can obtain it (by doing this we also find the value of d). Once we check whether we can get it, we add $2^{|B|-d}$. If $m = 0$ we actually add $2^{|B|-d} - 1$ because we don't want to count the empty set.

Now the question becomes how do we find the basis for each mask? Doing this naively can be done in $O(n \cdot \max a_i \cdot 15)$ or $O(\max a_i^2 \cdot 15)$, which are too slow. The trick is to iterate through all the masks, and find the new basis through old ones. Let's look at one ? of the mask. First we replace the ? with a 0 then a 1, to get to new masks mask_0 and mask_1 . Intuitively, it makes sense we can find the basis for mask when we know the bases for mask_0 and mask_1 . It can be tempting to do this by appending a 0 to the beginning of every vector in the basis mask_0 and a 1 to the beginning of every vector in the basis mask_1 and finding the basis of this set. This however won't give us a correct basis, since the 1 we appended at the beginning of the vectors for mask_1 can become a 0 during Gaussian elimination. The trick is that we actually know what this bit should be for each vector in the basis for mask_1 , because we already know what happened to a bit appended to always be 1 in the Gaussian elimination for mask_1 : it's the last bit we added to track parity! So actually, we append 0 to the beginning of every vector in the basis of mask_0 and the last entry (parity entry) to the beginning of every vector in the basis of mask_1 , and merge them using Gaussian elimination.

Case 2: $|s|$ is odd. This is in some sense the harder case, but we will use a lot of the same ideas. First to identify what the bits look like. Here it is more uncomfortable since if AND has some bit set, then so does XOR. So for each bit set in OR, it can be set zero, one or two times on the left side. Subtracting the right side of the equation from the left we get $\sum \pm 2^{i_k} = 0$, where all i_k are distinct, which is possible only if the sum is empty (because binary representation is unique). So the

only possible way is for each bit set in the OR to be set only in XOR i.e. each bit is either set nowhere or set in some odd number of elements but not all of them.

Now suppose we take a mask like last time, but now consisting only of 0 and ?. Let $B \subset A$ the elements matching this pattern restricted to the ? positions again. We need the number of subsets of odd subsets of B that xor to $(1, 1, 1 \dots, 1, 1)$ and no bit is contained in all of them. This second condition is impossible to model using linear algebra. Best we can do is find the total number of such subsets (appending by 1 again to ensure its parity) and try to take away the sets which contain bits that are set in every element.

The way we do this is the principle of inclusion-exclusion. Let's again take our ternary masks like in the previous case. We find the number of sets that have all zeros and ones on those positions, and has an odd number of ones on all ? positions. Then we add $(-1)^{(\# \text{ of } 1\text{s})} * 2^{|B|-d}$ to the solution if we pass the check. Let's see what happens to a set that doesn't have any bit set to 1 everywhere. It's counted just once, for the mask with only ? and 0. If a set has exactly k bits which are 1 everywhere, then it's counted 2^k times, but since we alternate signs, we actually count it $\sum(-1)^t \binom{k}{t} = 0$ times. So we counted everything the correct amount of times.

We now may solve this case in parallel with the first case. Once we obtain the basis for some mask, we check whether we can get $(1, 1, \dots, 1, 0)$ and if needed we can add the adequate amount from the first case to the solution, and then check whether we can get $(1, 1, \dots, 1, 1)$ and if needed we add the amount from the second case. The complexity of this solution is $O(3^k \cdot k^2)$ where $k = \log_2 \max a_i$ (more precisely $O(n + \max a_i^{1.58} \log(\max a_i)^2)$) with the only part actually having log squared being merging bases). However, when calculating how many operations we actually need (summing the number of ? squared for each mask) we get about $1.2 \cdot 10^8$, which runs very quickly, especially that we use the xor operation in the Gaussian elimination for merging bases. The rest of the solution works in $O(3^k \cdot k)$.

J. Make Swamp Great Again

Author: Petro Tarnavskyi
Solved by: 75/80
First to solve: 2Popici1Arici

Suppose we want to equalize the preferred temperature of all the swamp creatures to a target temperature x (the initial temperature of some of the creatures).

Let cnt be the number of creatures whose preferred temperature is different from x . To change all creatures' temperatures to x , we need at least cnt evenings. This is because, in one evening, we can change the temperature of at most one creature to x . Thus, each creature with a different temperature requires its own evening to change.

If there are at least two neighboring creatures with the target temperature x , we can use them to propagate x further. Specifically, for any neighbor of these two creatures, we can change their temperature to x in one evening. Using this, we can iterate clockwise around the circle:

- If a creature already has temperature x , we skip it.
- If a creature has a different temperature, we spend one evening changing it to x .

Initially, there might not be two neighbors with the temperature x . However, if we can find any group of three consecutive creatures where one creature's temperature can be changed to x , we can create two neighbors with x . After that, the propagation process is the same as mentioned above.

If there is no triplet where a creature's temperature can be changed to x , it means we need at least $cnt + 1$ evenings to change all temperatures to x . This extra evening is required to create a group of three consecutive creatures where one creature's temperature can be changed to x . Suppose we have a triplet of consecutive creatures with temperatures x, y, z , where (without loss of generality) $y \leq x \leq z$. In this case, we can change z to y . After this operation, we change y to x , creating two consecutive creatures with the target temperature x , enabling the propagation process to continue as before.

Thus, to solve the problem, we only need to determine if at least one group of three consecutive creatures where one creature's temperature can be changed to x . This determines whether the answer is cnt or $cnt + 1$.

Runtime: $O(n)$.

K. Intrusive Donkey

Author: Yarema Stiahar
Solved by: 32/41
First to solve: *Infinity*

Let a group be a sequence of identical characters that appear consecutively. Let's g_i be the size of i -th group. The relative order of such groups does not change, only their sizes.

To process a query of the first type, we need to find the first and the last group that will be changed by it. That process will be the same as a query of the second type. The first type of queries covers some groups entirely and at most two groups partially. Groups which are covered entirely will double in size. The group at both ends will increase in size by the number of their elements that are in range of a query. These modifications can be done with segment tree.

The second type of query has to find the smallest index j such that $\sum_{i=0}^j g_i \geq x$. It can be done with binary search or descent on Segment tree.

This problem can also be solved with Fenwick tree, as each group cannot double in size more than $O(\log A)$ times. This means that we can modify each group size naively.

Complexity: $O(n \log n)$ with descent and Segment tree or $O(n \log n \log A)$ with Fenwick tree.

L. Ogre Sort

Author: Roman Bilyi
 Vlad Ulmeanu

Solved by: 37/52
First to solve: FMI-1

We denote a move by (i, j) , $i \neq j$. If we perform the same move k times, we note it by $(i, j)^k$. Let $post$ represent the position at which we can find t in the permutation. We assume that we update pos after every move that we do.

Property 1: An optimal sequence of moves never contains $(i, j > i)$. A sequence of moves that achieves the same result, while only utilizing the $(i, j < i)$ kind of moves is $(j, 1)^{j-i}$, $(j-1, 1)^{i-1}$, but it only requires a cost of $j-1 < j$.

Property 2: $(i, j < i)$ can be replaced by a sequence of moves of the same cost of type $(i, 1)$: $(i, 1), (j, 1)^{j-1}$.

Following the first two properties, there must exist an optimal sequence of moves only of the type $(i, 1)$.

Property 3: Let $0 \leq t < n$ s.t. $post \not< post_{t+1} < \dots < pos_n$. If $t = 0$, the permutation is already sorted. Since $post \not< post_{t+1}$, the solution must contain at some time the move $(post_t, 1)$.

Property 4: If the sequence contains a move $(post_t, 1)$, then it must later contain $(post_{t-1}, 1), (post_{t-2}, 1), \dots (pos_1, 1)$ as well, because after $(post_t, 1)$ t will be in front of $1, 2, \dots t-1$.

Property 5: If the sequence contains a move $(post_t, 1)$, then the sequence mustn't later contain $(pos_{z>t}, 1)$, since t and z will be correctly ordered, and we can't break the correct ordering by doing any other moves required by property 4.

Using properties 3 and 4, the optimal sequence's length must be at least t , containing $(post_t, 1), (post_{t-1}, 1), \dots (pos_1, 1)$. Using property 5, we shouldn't use any other moves. There is a sequence of moves whose length is exactly t : $(post_t, 1), (post_{t-1}, 1), \dots (pos_1, 1)$ (using properties 4 and 5, we won't need to redo any moves if we employ this order).

We now have to figure out how to efficiently update pos .

Property 6: When we have to do $(pos_{1 \leq z \leq t}, 1)$, pos_z will be higher than its original value by the number of values $z < y \leq t$ that were originally to the right of z . We move them to the first position earlier, therefore each increasing z 's position by one.

We need a data structure that supports point updates (i.e. we moved a value from index i to the first position) and range sum queries (how many values that were originally after us were moved before us to the front). A Binary Indexed Tree is enough for this.

Although we can sort the permutation with a cost of $O(n)$, it's not easy to find a solution that generates the required moves faster than $O(n \log n)$.

M. Enchanted Lawns Quest

Author: Roman Bilyi
Solved by: 1/11
First to solve: *Infinity*

Let's make a binary search to find the answer. Now we need to check whether we can add weights such that the diameter is $\leq x$.

Let's call the middle point of the diameter C . It means that the distance from point C to any vertex is at most $x/2$. And the existence of such point also means that diameter is $\leq x$ – the distance between vertices a and b $dist(a, b) \leq dist(a, C) + dist(b, C) \leq x/2 + x/2 = x$.

The point C should lie on an edge (considering vertices also lie on an edge). Let's iterate an edge (a, b) where the point C lies. Let len be the original length of this edge.

Property: We should only add weights to edges which connect edges to their corresponding parents. If we add some positive amount to an edge not connecting a leaf, we can add the same value to all edges connecting leaves in the subtree instead.

Now let's find whether the point C can lie on an edge (a, b) . Let's call mx_a – maximum distance from vertex a to a leaf not using edge (a, b) . Same for mx_b . Firstly, $mx_a + len + mx_b$ should hold.

Let $0 \leq y \leq len$ be a distance from point a to C . If x is even, y should be integer. Otherwise, y is integer + 0.5 (we can multiply all distances by 2, then we have a condition whether y is even or odd).

The original distance from any leaf to C can't be more than $x/2$, so $mx_a + y \leq x/2$ and $mx_b + len - y \leq x/2$, so $y \in [max(0, mx_b + len - x/2), min(len, x/2 - mx_a)]$. If there is no possible solution, the edge (a, b) doesn't work.

Now let's count how much weight we can add for a fixed y and compare it with w . Let da_1, da_2, \dots, da_k be the distances from a to all leaves not using edge (a, b) . Same for db_1, db_2, \dots, db_m . We can add $\sum_{i=1}^k (x/2 - y - da_i) + \sum_{i=1}^m (x/2 - (len - y) - db_i) = (x/2 - y) \cdot k + (x/2 - (len - y)) \cdot m - \sum_{i=1}^k da_i - \sum_{i=1}^m db_i$. It's clear that the function is linear on y so we can try minimal and maximal possible values of y .

Now, how to compute all of this fast. It's enough to compute values $mx_a, cnt_a, sum_a, mx_b, cnt_b, sum_b$ for each edge.

Let cnt_a be the number of leaves reachable from a not using edge (a, b) .

Let sum_a be the sum of distances from a to all leaves not using edge (a, b) .

Then the formula become $(x/2 - y) \cdot cnt_a + (x/2 - (len - y)) \cdot cnt_b - sum_a - sum_b$.

Now we only need to find all of those values for each edge. This is quite standard tree problem. We can compute all of those values for one edge and then recompute how the values change when we want to compute them for incident edges. This can be implemented in $O(n)$.

The complexity of the whole solution is $O(n \cdot \log maxAnswer)$.