Farmula 1 (farmula1)

Author: Stefan Dascalescu

Developer: Stefan Dascalescu

Solution

We just have to count for the driver in question how many points they won using the rule system mentioned in the statement. Then, in order for another driver to put as good of a challenge as possible, they have to get the best possible result which was not obtained by our driver. If the driver in question won the race, then this other driver would get the second place. Otherwise, this driver would get the first place.

Last but not least, we only have to compare the two results and check whichever one of them was better.

Swapping Digits (`swappingdigits`)

Author: Bernard Ibrahimcha

Developer: Bernard Ibrahimcha

Solution

Observation 1: The integer $N > 0$ is a multiple of 25 if and only if it ends with 00, 25, 50, or 75.

Observation 2: If it's possible to make N divisible by 25, it's possible to do it using at most 2 moves.

The solution becomes as follows:

If N is already divisible by 25, then print 0.

Otherwise, check if swapping the last two digits with each other makes N divisible by 25, or if swapping one of them with another digit of N makes N divisible by 25, if so, then print 1.

Otherwise, check if there exists two 0s, a 2 and a 5, a 5 and a 0, or a 7 and a 5, if so print 2.

Otherwise, print -1.

All the above checks can be done easily by calculating the number of occurrences of each digit beforehand.

Precise Average 2 (avg2)

Author: Áron Noszály

Developer: Tommaso Dossi

Solution

Similarly to the first task, we will only work with the sums of the prices.

Obviously if we can change the prices in such a way that the maximal absolute change is at most C, we can also change the prices where it's at most $C + 1$. This suggests to binary search on the answer.

To test whether the answer is at most C, for every price P_i consider to what value it can change. It can be seen that the changed price is from the interval $[\max(1, P_i - C), P_i + C]$. The sum of prices thus is from the interval:

$$[L, R] = \left[\sum_{i=1}^{N} \max(1, P_i - C), \sum_{i=1}^{N} P_i + C\right]$$

If $N \cdot K \in [L, R]$ the answer is at most C.

The time complexity is $O(N \log(\max P))$. The task can also be solved with sorting in $O(N \log N)$, which is left as an exercise for the reader.

String Imbalance (`stringimbalance`)

Author: Alexandru Lorintz

Developer: Alexandru Lorintz

Solution

Let's try to give more meaning to the value we want to compute. Recall that the imbalance of a string is just the number of unordered pairs of characters from it which are different. If we want to minimize this using the given operations, we may also think of maximizing the number of pairs of equal characters, as their sum is constant $\frac{N(N-1)}{2}$, where N is the length of the string).

With this in mind, we can observe that a greedy strategy for solving the problem actually works. Let's consider some state of the string and we want to make only one operation in order to maximize the pairs of equal characters. It's always optimal to change one of the least frequent characters (which appear the least in the string) to one of the most frequent. This way, when changing one of the least frequent characters, we will "loose" the least number of equal pairs and later "gain" back the most possible by changing in the described way.

In order to compute the minimum imbalance we will always need the list of sorted frequencies of characters, but this is easy to maintain by always propagating he newly updated character to the right of the list if necessary.

The time complexity for solving an operation is $O(\Sigma)$, where Σ is the number of letters in the alphabet (52 in our case), so for the whole solution it is $O(Q \cdot \Sigma)$, but a solution that sorts the frequency array each time for a $O(Q \cdot \Sigma \cdot \log(\Sigma))$ time complexity should also get a full score.

Tulip Bouquets (`tulips`)

Author: Áron Noszály

Developer: Áron Noszály

Solution

Consider the straightforward DP solution: let $F(i, j)$ be the maximum sum of beautinesses if we create j bouquets from the first i tulips. With obvious base cases, the transition is:

$$F(i,j) = \max_{k=0}^{i-1} \left(\min_{j=k+1}^{i} T_i + F(k, j-1) \right)$$

This takes $O(N^2K)$ time to calculate. To optimize, calculate F in this order: $F(*,1), F(*,2), \ldots, F(*,K)$. While calculating F values with a fixed j, loop i from 1 to N and maintain a stack of pairs in the form of: $(\min_{j=k+1}^{i} T_i, F(k, j-1))$. This stack can be updated in amortized $O(N)$ time. To calculate $F(i,j)$ in $O(1)$, we also need to maintain a maximum stack[1] from the sum of these pairs.

The time complexity is $O(NK)$.

[1] consult cp-algorithms.com

Good Intervals (`intervals`)

Author: Alexandru Lorintz

Developer: Alexandru Lorintz

Solution

The crucial observation for solving the problem is that in order for an interval to have its right end in some value, its length should divide it without a remainder, so there are $O(N \cdot \log(10^{18}))$ good intervals, as the lcm (least common multiple) of all the lengths that end in a value should divide it without a remainder and also the values before should be divisible by their certain index values.

With that fact in mind, we can just find all the intervals and do a simple sweep-line algorithm (exploiting the fact that we can solve the queries "off-line") using a binary indexed tree for finding all the intervals completely included in a given query interval.

The overall time complexity is $O(N \cdot \log(10^{18}) \cdot \log(N))$, where the $\log(N)$ comes from the binary indexed tree update time.

Increasing XOR (`increasingxor`)

Author: Péter Gyimesi

Developer: Bence Deák

Solution

Denote the position of the most significant set bit of a positive number k as $\beta(k)$ (e.g. $\beta(19) = 4$). For an array B, denote the array of its prefix-XORs as $\text{pre}(B)$. Let $L \leq 29$ be the maximum possible set bit, and for all $p = 0, \ldots, L$, define

$$S_p(B) := \{i : \beta(B_i) = p\},\ T_p(B) := \{i : \text{the } p\text{th bit is set in } B_i\} \setminus S_p(B), \Delta_p(B) := |T_p(B)| - |S_p(B)|$$

Observation: $x < x \oplus y \Leftrightarrow$ the $\beta(y)$th bit is not set in x. Therefore, if $P = \text{pre}(C)$ for some array C of size k, then $P_0 < \cdots < P_{k-1} \Leftrightarrow$ the $\beta(C_{i+1})$th bit is not set in P_i for all $i = 0, \ldots, k-2$.

Claim: The array B is beautiful iff $\Delta_p(B) \geq -1$ for all $p = 0, \ldots, L$.

$\Rightarrow$

Let C be an arbitrary permutation of B, and $P := \text{pre}(C)$. Obviously, $\Delta_p(B) < -1 \Leftrightarrow \Delta_p(C) < -1$.

If $\Delta_p(C) < -1$, then – according to the pigeonhole principle – there exist two indices $l < r$ such that $l, r \in S_p(C)$, and the pth bit is not set in C_i for all $l < i < r$.

The pth bit changes exactly twice in the subarray $P[l-1], P[l], \ldots, P[r]$; at index l and at index r. Since $\beta(C_l) = \beta(C_r) = p$, according to our observation, either $P_{l-1} > P_l$ or $P_{r-1} > P_r$, so B cannot be beautiful.

$\Leftarrow$

Let's build a solution inductively, starting with the elements whose most significant set bit is the greatest. Suppose we already have an array C containing the elements $\{B_i : i \in S_{p+1}(B) \cup \cdots \cup S_L(B)\}$ such that $\text{pre}(C)$ is strictly increasing.

If $\Delta_p(B) \geq -1$, we can insert $H := \{B_i : i \in S_p(B)\}$ into C in the following way:

1. Insert the first element of H before C_0.
2. Insert the $(i+1)$th element of H directly after the ith such element of C whose pth bit is set.

Let C' be the resulting array, and $P := \text{pre}(C')$. It's easy to see that for all $i = 0, \ldots, |C'| - 2$, the $\beta(C'_{i+1})$th bit is not set in P_i, therefore P is strictly increasing.

Algorithm: We iterate over the indices $0, \ldots, N-1$ while maintaining an array `delta` of size $L+1$, initialized to $[0, \ldots, 0]$.

At index i, we decrement `delta`$[\beta(A_i)]$ and increment `delta`$[p]$ for all $p < \beta(A_i)$ such that the pth bit is set in A_i. The array $[A_0, \ldots, A_i]$ is beautiful iff `delta`$[p] \geq -1$ for all $p = 0, \ldots, L$.

Pac-Man (pacman)

Author: Filippo Casarin

Developer: Filippo Casarin

Solution

Given G a set of open cell, let's say that G is *greedy* iff the answer to the problem is `YES`.

Given a set S we say that S is a *slice* of G iff there exists an axis-aligned subspace T (i.e. $\{z = z_0\}$) such that $S = G \cap T$. Notice that G is a *slice* of itself.

Lemma 1: G is *greedy* $\implies$ every *slice* of G is connected.

Assume there exists a *slice* S containing two disconnected points A and B, a ghost can't reach B starting from A since following Alessandro's strategy a ghost will never try to exit S and there's no path in S, thus we have a contradiction.

Lemma 2: Every *slice* of G is connected $\implies$ G is *greedy*.

Assume G is not *greedy*. Consider S a non-*greedy* *slice* of G not containing any non-*greedy* *slice*.

Since S is not *greedy* in contains two points A and B such that a ghost moving from A to B will get stuck at point C.

Consider the shortest path $P_0, P_1, \ldots, P_k$ contained in S from C to B, let's assume that the first move $P_0 \to P_1$ is along the x axis, notice that the distance along this axis will increase. Since we have to reach the point B eventually the distance along the x axis of some point P_i and C will be the same.

Consider $S' = S \cap \{x = C_x\}$, S' is *greedy* by construction of S. A ghost following Alessandro's strategy will find a path from C to P_i and this path will be shorter than the original path $P_0, P_1, \ldots, P_i$ contradicting the minimality of path P.

Algorithm
We just need to check that every *slice* of G is connected, an element of G is part of at most 7 *slices*, the time taken to test the connectivity of a *slice* is linear in the number of elements it contains, so the total time is $O(7n) = O(n)$.

The Jeweller's Game (gemgame)

Author: Áron Noszály

Developer: Áron Noszály

Solution

Let's denote by $V(g)$ the value of group g and by $G(i,j)$ the group of cell (i,j).

Without loss of generality, let's only consider horizontal swaps and try to calculate the value of one of the swapped cells. So we want to calculate the value of $(i,j+1)$ after swapping (i,j) and $(i,j+1)$. A straightforward solution is to perform a depth first search on the board after the swap, but overall this takes $O(N^4)$ time: only enough for the first two subtasks.

A naive way to optimize the previous calculation would be look at cells $(i+1,j+1),(i-1,j+1),(i,j+2)$ (assuming they exist and have the same gem type) in the original board and say that the value of $(i,j+1)$ is:

$$y + \sum_{x \in \{G(i+1,j+1), G(i-1,j+1), G(i,j+2)\}} V(x)$$

where y is 1 if $G(i,j) \notin \{G(i+1,j+1), G(i-1,j+1), G(i,j+2)\}$ and 0 otherwise.

But this has huge flaw, consider this case (swapping the blue numbers):

3	3	1	3		3	3	**1**	3
1	**1**	**2**	1	⇒	1	2	**1**	**1**
3	1	1	1		3	**1**	**1**	**1**

The problem is: after the swap the one on the left gets disconnected from the rest, because (i,j) was a articulation point. So in the correct solution, if (i,j) is an articulation point we must consider the connected components in the original board after removing (i,j) and only adding those that will affect the answer.

It's a bit tricky to implement, but a time complexity of $O(N^2)$ can be achieved.