Tuscan Peaks (peaks)

Author: Tommaso Dossi

Developer: Tommaso Dossi

Solution

Picking one cell, we can look at the cells next to it. (One array contains a, b, c and the other x, y, z.)

$a \cdot z$	$b \cdot z$	$c \cdot z$
$a \cdot y$	$b \cdot y$	$c \cdot y$
$a \cdot x$	$b \cdot x$	$c \cdot x$

$b \cdot y$ is a peak iff

$$ay < by > cy$$
$$bx < by > bz$$

Since now everything is positive we can divide by y and b:

$$a < b > c$$
$$x < y > z$$

This means that by is a peak iff b is peak and y is a peak in the corresponding array. So we only need to find the number of peeks in the given arrays, and the answer is the product. Finding peaks in the given arrays can be done in $O(N + M)$ time.

Duplicated Usernames (usernames2)

Author: Alessandro Bortolin

Developer:

Solution

There are several ways of solving this problem, all of them relying on sorting the strings in some way or another. Probably the easiest approach is to find which strings have the prefix equal to the string we are given and then we can keep either a frequency array or a set to store the numbers resulted and then all we have to do is to check what is the smallest number that doesn't show up in the preferred data structure, thing which can be done using a simple traversal.

Another idea is to add all strings in a map and start generating potential answers, starting with the one which doesn't have any digit and then you can check for each of these strings if they exist in the map or not. The algorithm would run until you find a string which doesn't exist in the map, and all you have to do is to find the required string.

Largest Rectangle (rectangle)

Author: László Nikházy

Developer: László Nikházy

Solution

We need two pairs of equal-length sticks to form a rectangle since the opposite sides of the rectangle have the same length. These pairs of sticks should have the maximal possible length. There are multiple possible approaches to solve the problem efficiently, the easiest implementation uses sorting.

Let us sort the array of stick lengths in decreasing order because then the equal elements will be next to each other and the largest elements will be in the beginning. We search for the first two consecutive equal elements, which will be one pair of sides for the rectangle and then we continue the search afterwards again searching for the next two consecutive equal elements, which will be the other pair of sides. If we cannot find two pairs of equal elements then there is no solution. The complexity is $O(n \cdot \log n)$ because of the sorting.

Another option is to count occurrences of each element using a C++ map or Python dict, or even a simple array that is as large as the maximal value (which was at most 10^6). With this approach, we need some additional casework. There are two major cases: the largest element with multiple occurrences might occur at least 4 times, and then we should select 4 of that element, or if it occurs 2 or 3 times, we need the second largest element with at least 2 occurrences. The complexity is $O(n + \max S_i)$ if we use a simple array.

Swapping Brackets (bracketswap)

Author: Áron Noszály

Developer: Áron Noszály

Solution

The following greedy approach works: swap the first occurrence of ")" with the last occurrence of "(" until the sequence is balanced. This can be implemented in $O(N^2)$ naively. To optimize this approach let's think in terms of the usual modifications when dealing with balanced bracket sequences: replace the opening ones with $+1$s and the closing one with -1s and consider the prefix sums. The sequence is not balanced while there's a negative prefix sum. Notice that the swap we do essentially increases the *current* minimum prefix sum by 2 (though we might think the minimum prefix sum after the swap might be at a different location, but it turns out, it can only happen after the the sequence becomes balanced and even then, it may only change to the last position of the sequence).

Now our solution is just maintaining the minimum prefix sum and searching for the first ")" and last "(" in a two-pointer like fashion. The time complexity is linear.

Excellent Numbers 2 (excellent2)

Author: Péter Gyimesi

Developer: Stefan Dascalescu

Solution

In order to solve the problem we can start with a simple solution that involves combinatorics or DP. One such approach would be to use DP where $dp[i][j]$ is the number of integers with i digits which only have 1 and 5 and their reminder when dividing by 3 is j. From $dp_{(i,j)}$ we can proceed towards $dp_{(i+1,(j+1) \mod 3)}$ or $dp_{(i+1,(j+2) \mod 3)}$ depending on adding a 1 or a 5. The solution would be found in $dp_{(n,0)}$. However, this solution is $O(n)$ and it is too slow.

However, we can optimize this approach by analyzing the answers we can get for the first few values of n and there are two possible approaches from here. The first one is to find that if we note with dp_n the answer for a given n, it is equal to $dp_{n-1} + 2 \cdot dp_{n-2}$, which allows us to now solve the recurrence with a rather standard matrix exponentiation, similar to finding the k_{th} Fibonacci integer.

There is also another approach, a pure mathematical one, which reduces to finding closed form formulas for odd and even n, there are multiple potential formulas which work here but one of them is $\frac{2^n-2}{3}$ for odd n and $\frac{2^n+2}{3}$ for even n.

Both the matrix exponentiation as well as the closed form formula can be done in $O(\log n)$ time.

Remark: The recursion $dp_{n+2} = 2^n + dp_n$ can be used to get the above closed form using only the geometric sequence sum formula.

It is easy to see that $\begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} * & * \\ * & dp_n \end{pmatrix}$.

Dog Trick Competition (dogtrick)

Author: Zsolt Németh

Developer: Bogdan Ioan Popa, Casian Patrascanu

Solution

The first important observation is to notice that greedily performing the tricks does not work. For example, suppose that the dog knows tricks 1, 2, and 3, together with the transitions $1 \to 2$, $1 \to 3$, $2 \to 2$, $3 \to 3$. Now, for the trick list $1, 2, 3, 2, 3, 3$ the greedy solution yields 0 points (performing tricks 1, 2, 2 and then failing to continue), while the optimal solution gives 4 points (performing tricks 1, 3, 3, and 3).

Therefore, we will utilize dynamic programming. Note that when evaluating T_i (the i-th trick on the list), we may compute the maximum score that we can achieve *starting from* trick T_i by the following recursion:

$$DP_i = \max(\, tr(T_i, T_{i+1}) \cdot DP_{i+1},\, tr(T_i, T_{i+2}) \cdot DP_{i+2}),$$

where $tr(x, y)$ is 1 iff trick y can be performed right after trick x, otherwise it is 0.

We could implement this as a simple recursion, but it is too slow to score high. The other observation we should make is that we can utilize memorization by evaluating the list backward: first, check the last two tricks on the list and calculate their score value, then iterate towards the start of the list and compute scores using the above formula. To report the correct result, we must be careful and consider both T_0 and T_1 as the potential first trick.

Implementation-wise, we may store the DP_i scores in a simple array. A possible method to find the required $tr(x, y)$ values efficiently is to store the transitions in an array of sets, where the i-th set contains the tricks that we can perform right after trick i. Note that we may reduce the amount of casework to process the beginning and the end of the list by appending "dummy" tricks that allow all possible transitions.

Stefan's New Year's resolutions (manymax)

Author: Stefan Dascalescu, Ovidiu Rata

Developer: Stefan Dascalescu

Solution

This is a fairly typical data structure problem where we can use multiple solutions in order to make it work. For the first few subtasks, a variety of solutions including storing prefix products, brute forcing each value as well as storing a segment tree for maximums are all viable options towards solving these smaller subtasks.

In order to solve the full problem, we need to be able to answer to the query of finding the k_{th} maximum fast enough. The first approach would be to build a merge sort tree (segment tree where the entire array is stored) and then for each query, have a binary search where for each node we find how many values are greater or equal to a certain value. Unfortunately, the complexity of the query if done in this way would be $O(\log^3 N)$ which is too slow. However, we can use this insight to speed up our algorithm.

What if instead of solving a query at a time, we want to solve everything at once? Actually, this is possible and this allows us to use parallel binary search in order to answer to each query. This is possible because we are allowed to answer to queries offline and now all we have to do is to sort the queries about $\log n$ times and at each step, we try to find if for each query, the k_{th} maximum is greater or equal than the value we are seeking to achieve. This is now doable in a very easy manner using a fenwick tree or a segment tree, the only thing we will have to consider is making sure to avoid overflows and to compute the mod operations properly as the time limit is rather tight and the mod operations can be very costly. The final complexity of the query will be $O(\log^2 N)$ which is good enough for this problem.

There are however alternative solutions that also solve the problem, using either more powerful data structures such as persistent segment trees, as well as solutions that run in $O(\sqrt{n})$ per query after running Mo's algorithm and optimizing the process of computing the answer for queries using another square root decomposition.

Carry Bit (carry)

Author: Filippo Casarin

Developer:

Solution

First observe that whenever we have two bit sequences of length L: it is easy to decide that their sum fits in L bits or not. Simply find the first position from the left, where they are the same. If these are 1, then we have overflow. Otherwise (0) independent from the rest of the digits: their sum fits in L digits. (This can be seen below, where $\bar{x} = 1 - x$.) If the ith bits are different for all index, then the sum is $11\ldots1$: fits in L digits.

$x_0x_1\ldots x_k1\ldots$	$x_0x_1\ldots x_k0\ldots$
$\bar{x}_0\bar{x}_1\ldots\bar{x}_k1\ldots$	$\bar{x}_0\bar{x}_1\ldots\bar{x}_k0\ldots$
overflow	sum fits in L bits

So for each query we only need to find the first equal bits from the left. Instead of this we invert one of the bit-sequences, and now we need to find the length of the matching sequence from the left. We can do this by hashing[1] and binary search. The prefix hash values can be computed in $O(N)$ time, and answering all queries cost $O(Q \log N)$.

[1]For details of hashing: look at the solution of `periodicwords` from the first round.

Strange Operation (strangeoperation)

Author: Péter Gyimesi

Developer: Bence Deák

Solution

Define the function s (which maps an N-sized array to an $(N-1)$-sized array) as follows:

$$s(X)[i] := p(i) \cdot (X[i+1] + X[i]) \ (i = 0, \ldots, N-2), \text{ where } p(i) := \begin{cases} 1 & \text{if } i \equiv 0 \pmod 2 \\ -1 & \text{if } i \equiv 1 \pmod 2 \end{cases}$$

One can see that performing the operation at index $i+1$ corresponds to swapping $s(A)[i]$ and $s(A)[i+1]$. Since knowing $A[0]$ and $s(A)$ is sufficient to reconstruct the rest of the elements of A, we only have to check the validity of $s(B)$, that is, whether $s(B)$ is a permutation of $s(A)$. In other words, a necessary and sufficient condition for the validity of B is:

$$A[0] = B[0] \text{ and } s(A) \text{ is a permutation of } s(B)$$

This can be checked easily in $\mathcal{O}(N \log N)$ (e.g. by sorting or using std::map).

The minimum number of swaps required to transform $s(A)$ into $s(B)$ is a known problem. It requires computing the inversion count of the permutation, which can be done in $\mathcal{O}(N \log N)$ (e.g. using a segment tree / Fenwick tree / order statistics tree or using merge sort). The only difficulty is – since there are repetitions in the arrays – that the permutation that transforms $s(A)$ into $s(B)$ is not unique. However, it's easy to see that if we preserve the relative order of equal elements, the inversion count will be minimal.