USACO.Guide Informatics Tournament

March 24, 2024

1 TriNum Array

The solution to this problem lies in the implementation of the sliding window technique using two pointers (i and j) to define a window within the array. We can use a map m to keep track of the frequency of elements within the window.

The algorithm incrementally expands the window (j) by updating the frequency in the map m. If the window violates the constraint of having at most three different numbers, it shrinks the window from the left side (i) until the constraint is satisfied again.

At each step, update the ans variable with the maximum length of the subarray meeting the conditions ($j - i + 1$).

Finally, output the length of the longest subarray meeting the specified conditions, represented by the ans variable.

2 Two Way Homework

Solution 1:

The main idea of this problem relies on the fact that if we use the value from the first array for way too many times, the costs will become so big that any benefit we might get from using that array would go away.

Therefore, we know that we will only be able to use the values from the first array at most 40 times or so. Now, we turned this into a straightforward DP where $dp(i, j)$ is the minimum answer if we used i values from the first array by the time we arrived to position j.

The transitions are pretty typical as we can go to either $dp(i + 1, j + 1)$ or $dp(i, j + 1)$.

Solution 2:

Alternatively, we could just iterate over the array in reverse order from $i = n$ to 1 simulating the process backwards and keep track of the answer for the suffix. At every index i, we can either add B_i to our answer for the suffix or double the answer for the suffix and add A_i to it.

3 Balanced Difference

Hint: The minimum and maximum number of the array are always included in the calculation. No matter what partition we choose.

Solution: Useful Observation: We can see that, if we choose any partition, the minimum and the maximum element must be present in the difference calculation of each partition.

What does this lead to? Let's call the max and min element a and b respectively. Now we require two numbers, say x and y, that satisfy the following condition

$$x - a == b - y$$

Rearranging the equation, we get

$$x + y = a + b$$

We know what a and b are, what's left is finding x and y. And we can do that using classical two-sum problem.

4 Producing Digits

First note that the brute force solution is extremely slow, as you have to check every single possible subset for each index. This leads to an exponential time complexity.

One observation we can make is that only the unit digit matters, since none of the other digits can affect the unit digit of the product. Also remember that the numbers can be negative, so make sure to use an absolute value when getting the units digit.

Another observation we can make is that once we can construct a unit digit once, it can always be reconstructed in the future to multiply. Using this, we can keep a boolean array of size 10 which keeps track of which digits are possible to construct and which digits are not. From there, we just check if multiplying the unit digit of the a_i by any of the previous possible unit digits creates b_i. We then update the boolean array with any new possible constructed digits before moving onto the next index. This solves the problem in linear time.

5 Gardening is Hard

Because there are no vertically adjacent wells, the closest well will always be the closest i where there is a well in column i. This allows us to simplify the problem to something similar to the solution if a strip was 1 row instead of 2. Because of this, we can simply loop through all possible strip rows, and loop through all the right endpoints of possible strips. Then we can count the number of valid strips with a specific endpoint by maintaining the last well and the last well answer (because all right endpoints that connect to the same well will have the same answer). Then we can update the answer when we reach a new well, by checking if the two wells are connected. This requires the knowledge of both rows of the wells, and with this, you can create a check to see if two wells are connected (wells being connected in this case means that all locations in between them are close enough to either well).

6 Farmer John's Cities

It is obvious that we should create a graph where an already existing road from u to v corresponds to an edge.
It is also clear that we have to use Dijkstra's Algorithm.
We can see that the most logical solution would be the following:
Take each pair (a, b) of the K new roads that can be built, and find the minimum distance from s to a and from b to t. Let them be d_1 and d_2 respectively. Then the answer for that road (a, b) would be $d_1 + L(a, b) + d_2$.
Finding the minimum distance from s to a for each pair of new roads (a, b) is easy, since we can easily just run one Dijkstra from node s and then in $O(1)$ get the minimum distance to any other node. However, how can we find the minimum distance from b to t fast enough for each of these pairs (a, b) ? Obviously, doing a Dijkstra for each node b each time is not optimal.
The trick is to reverse the direction of all the edges of the graph and run just one Dijkstra from t and find the minimum distance from t to b. Proof: Suppose the shortest path from b to t was the following:

$b \to u_1 \to u_2 \to \cdots \to u_k \to t$. Now, we reverse all the edges in the graph. For the sake of contradiction, we assume that the path $t \to u_k \to u_{k-1} \to \cdots \to u_1 \to b$ isn't the shortest one. Then, this means that a different path $t \to s_m \to s_{m-1} \to \cdots \to s_1 \to b$ is the shortest one. However, this also means that after reversing the edges back to their original direction the path $b \to s_1 \to s_2 \to \cdots \to s_m \to t$ would be the shortest path from b to t which is a contradiction.
Therefore, we only need to run a nomral Dijkstra and another one on the reversed edges and then compare the minimum overall of these distances we compute with minimum distance from s to t in the original graph (i.e. if we do not take any of the new roads).

7 Soccer League

In order to solve this problem you need to be careful about dealing with the various cases that the problem might involve.

The main idea behind getting a working solution for this problem is that you want to simplify the input as much as possible. First, ignore the obvious cases where there is no solution as there is no point in having to deal with them.

Then, we want to start from assuming we have $1 - 0$ and $0 - 1$ results for wins, respectively $0 - 0$ for draws. Depending on how many goals we are left, we can then try to see if there is one way of assigning the results, at least two or in some cases, none.

In order to explore all of the cases, I recommend seeing an official implementation.

8 Cheating the Group System

Hint 1: Think of the friendship / best friendship as a graph.

Hint 2: The only thing that matters are the connected component sizes

Proof Hint 2: You can pick any spanning tree of a connected component that is sufficient to generate any permutation of cards in that connected component.

Hint 3: Solve the problem if the best friendship graph was just 1 connected component.

Solution Hint 3: Let s = size of the connected component. Now we will add to the connected component until it is guaranteed that there exists s cards of the same type in the connected component. This is just a pigeonhole theorem and you get: $n(s-1)+1$ as the desired connected component size ($n(s-1)$ gives each card $s-1$ duplicates, and then we need 1 extra to ensure a there exists a card with s duplicates). This gives us that the answer is $n(s-1)+1-s$.

Hint 4: It is optimal to connect all connected components into one single connected component.

Proof Hint 4: Let's say we had two connected components with sizes s_1 and s_2. If we split them up, we get that the new connected components need to have sizes: $n(s_1-1)+1$ and $n(s_2-1)+1$ respectively. But after moving all of the same s_1 cards to the s_1 connected component, we still have $n(s_1-1)+1-s_1$ remaining extra cards, which if connected with the s_2 connected component, could reduce the necessary size of the new s_2 connected component. This is always the case because it costs 1 edge to connect the two connected components, and we gain $(n-1)(s_1-1)$ ($n \geq 2$ and $s_1 \geq 2$ so this is always ≥ 1) possible nodes to trade with.

Hint 5: To check if a S works we can do this: loop through each connected component (in some order), and let that size be s. $S \geq n(s-1)+1$. Then we do S -= s, as those s students' cards cannot be traded anymore as they are in the correct place (a similar idea to the proof of Hint 4). But what is the optimal order to minimize S?

Answer Hint 5: The optimal ordering is largest to smallest (intuitively this makes sense).

Proof Hint 5: Computing the min S is simple, just reverse the checking S algorithm. let s_i = the size of each connected component, let p_i = the sum of s_j where $j < i$. Let $v_i = n(s_i - 1) + 1 + p_i$. The answer is just max(v_i for all i). Now let's analyze what a swap of a and $a + 1$ does:

Only a and $a + 1$ are affected by this swap. If $s_a < s_{a+1}$,
$v_{a+1} = n(s_{a+1} - 1) + 1 + p_{a+1} \geq v_a = n(s_a - 1) + 1 + p_a = v_a$. This means the max v_a and v_{a+1} is v_{a+1} if $s_a < s_{a+1}$. Now after we swap we can prove that both new v's are less or equal to the old v_{a+1}:
$n(s_{a+1} - 1) + 1 + p_{a+1} \geq n(s_{a+1} - 1) + 1 + p_a$ and $n(s_{a+1} - 1) + 1 + p_{a+1} \geq n(s_a - 1) + 1 + p_a + s_{a+1}$. Because an unsorted array will always contain such an index, an unsorted array is never better than a sorted array, and so the sorted array is optimal.

Editorial: Read all hints first. Use a simple dfs to compute all the connected component sizes, and store those ≥ 2. Then sort those sizes and compute min S by computed v_i for each index and compute the max v_i. To finally compute the answer, firstly we add all edges to connect all connected components (which is just # connected components - 1), then add to S - the sum of all the sizes of connected components as one edge always increases the connected component size by 1. The final answer is just S + # connected components - 1 - sum of sizes of connected components.

9 Hori and Cake

First, notice that we can represent these segments as a tree. Then we have to answer the following question: How many sequences are there to remove some nodes (and when we remove a node we remove it's entire subtree) from a tree to completely delete the tree?

We can do an n^3 dp where $dp[i][j]$ = number of sequences to delete exactly j nodes for the ith subtree.

We can make transitions from the subtrees.

Let's say we have two nodes u and v where u is the parent of v.

Let's say that we have a sequence of x deletions in the subtree of v and a sequence of y deletions in the subtree of u none of which are in the subtree of v.

The number of ways to create a sequence of $x + y$ deletions that mixes all of these together is

$$\binom{x+y}{x = \frac{(x+y)!}{x!y!}}$$

.

We can extend this to multiple children. Let's say u has s children. If we have a sequence of $a_{S_1}, a_{S_2}, ..a_{S_s}$ where a_{S_i} is the length of the sequence of deletions in the S_ith subtree. The number of ways to merge all of these sequences together is

$$\frac{(\sum_{i=1}^{s} a_{S_i})!}{\prod_{i=1}^{s} a_{S_i}!}$$

The number of ways to mix all these sequences together while counting the coefficients of the number of ways to reach each of these sequence subtree sizes is

$$\frac{(\sum_{i=1}^{s} a_{S_i})! \cdot \prod_{i=1}^{s} dp[S_i][a_i]}{\prod_{i=1}^{s} a_{S_i}!}$$

These transitions can be computed with a knapsack like dp where we keep track of the sums of products of $\frac{dp[S_i][a_i]}{a_{S_i}!}$ and multiply $dp[u][i]$ by $i!$ after computing transitions from all children.

After computing these transitions, we simultaneously set $dp[u][i] = dp[u][i] + dp[u][i-1]$ because we can extend each sequence by 1 by deleting node u.

The only special case is where u is equal to the root of the tree. Depending on how we construct the tree, this node may not even be a segment of the tree and just an auxiliary node that is used to connect all independent subtrees. In this case, we need to modify the dp slightly in order to make sure all of it's children's subtrees are completely deleted.

We can speed this up to $O(n^2)$ by using the subtree merging trick (https://usaco.guide/adv/comb-sub) by running the dp while bounding the subtree sizes.

10 CRP Game

Let's make some useful observations first.

The numbers are 'pushed' in the same order as they were initially in board A.

Proof: No move changes the order in which the initial numbers appear in board A. Thus, the numbers are pushed in the order they were given. For any given initial sequence of numbers in board A, a solution always exists.

Proof: If the numbers in the sequence are 0 or 1 then we can make the sequence $000\ldots0111\ldots.1$ by doing a $'p'$ move anytime we meet number 1 and by doing $'rpr'$ each time we meet number 0.

Now, since we have numbers from 0 up to 3, we can do the following: Each time we meet number 2 we can do $'cpc'$ and each time we meet number 3 we can do $'crprc'$. Therefore, a solution always exists

It is not beneficial to do two $'c'$ moves without doing a $'p'$ in between.

Proof: By doing two $'c'$ moves without a $'p'$ in between, we do not affect any element that needs to be pushed. Since we are lookin for the shortest sequence of moves, it is not beneficial.

It is not beneficial to do two $'r'$ moves without doing a $'p'$ in between.

Proof: Like before, by doing two $'r'$ moves without a $'p'$ in between, we do not affect any element that needs to be pushed. Since we are lookin for the shortest sequence of moves, it is not beneficial.

The optimal answer will never contain $'rcp'$.

Proof: The $'r'$ move affects elements in board B, whereas, the $'c'$ move affects elements in board A. Therefore, we could do $'crp'$ instead of $'rcp'$, which is more optimal since $'crp'$ is lexicographically smaller than $'rcp'$.

Based on these observations we can reduce the initial problem, to the following:

We are given N numbers, and for each number we choose one of the following moves $'p'$,$'rp'$,$'cp'$,$'crp'$ such that we end up with N numbers in board B which are grouped (i.e. all 0s are together, all 1s are together etc) and additionally the sequence we chose should have the smallest length and be lexicographically the smallest one, in case of equality in length.

The whole idea for the optimal solution is based on Dynamic Programming with Bitmasks.

If the numbers in board B are already grouped correctly, then we notice that a number x_i can be inserted only if:

The rightmost number r in board B is x_i (i.e $r = x_i$ or $r = 3 - x_i$ in case the numbers in board A are complemented) x_i does not exist in board B (or $3 - x_i$ in case the numbers in board A are complemented)

We will take the case where the numbers in board A are not complemented, since either way we can use the same

arguments.

Proof: Obviously, if x_i is the rightmost number in board B, then we can just push it there, preserving the 'grouped' property. Otherwise, if x_i does not exist at all in board B, then we can just push it and create a new group for it, again preserving the 'grouped' property. In any other case, it is evident that pushing it breaks the 'group' property.

Knowing that, we can do the complement and reverse moves. We can make a state s = (position, is_complemented), where position[x] is the position of the group, number x belongs to (or 0 if x has not yet appeared), and is_complemented is a boolean variable which tells us if the numbers in board A are complemented or not.

We will now do dynamic programming in dp(i,s) which will give us the optimal answer when we are processing the i-th number, and we are at state s.

Furthermore, we will also use forward dp, where the transitions are the possible moves $'p', 'rp', 'cp', 'crp'$.

For simplicity purposes, for each state s we can also keep track of the following:

The positions array which we mentioned earlier The is_complemented boolean variable which we mentioned earlier The leftmost and rightmost values The count of the distinct elements that have appeared so far

We only keep the possible states for i and we compute the possible ones for $i + 1$. Also, in order to compare faster, we can store the values in a map where the key is the state and the value is the optimal sequence of moves that leads us there.

We obviously store that in a string, so that we can easily compare two strings and find the lexicographically smallest one. However, this makes it a bit slow, so later we will see an optimization to avoid this.

By implementing this correctly we can see that it leads to an $O(N^2)$ complexity which is slower than what we need.

Let's see how to make it faster.

The idea is to store the possible states for i, sorted in a lexicographical order of the optimal sequences of moves (the solutions) that we can do to reach that state. After, we check the moves for each of them in a lexicographical order i.e $'cp', 'crp', 'p', 'rp'$.

Like before, for each state we store the sequence of moves with the smallest length, and the first one (which is the lexicographically smallest one here) in case of equality of lengths.

Finally, we store the states of $i + 1$ based on the order of the sequences of moves that led to them.

We also keep track of a parent array par, where par[s] stores the moves that led us to state s. By backtracking on par we can retrieve the optimal answer.

This has $O(N)$ total time complexity and almost passes all test cases.

We need to do some bitmask tricks, since comparing and creating states take time.

We can use the fact that positions, leftmost, rightmost, and count store integers less than or equal to 5. This means we use only 21 bits of 7 integers. Since we only need 22 bits (21 + 1 for the is_complemented) we can store everything into a 32-bit integer.

Now, to optimize the complexity, we can use the following tricks:

Reading three bits from an integer (where _111=7)
Changing three bits in an integer
Reading one bit
Change of one bit

You can read about all of these here:

https://www.topcoder.com/community/competitive-programming/tutorials/a-bit-of-fun-fun-with-bits/.

You can see the code in the official implementations.

11 Counting Pairs

Note that the process is very similar to the Euclidean algorithm but in reverse. Because of this and the fact that we start from $(1, 1)$ the only pairs of numbers we can reach are coprime numbers.

Proof: We will prove this by induction. Let's say we can reach (x, y) and its gcd is 1. Then if we do $(x + y, y)$, then $\gcd(x + y, y) = \gcd(x, y)$, and if we do $(x, x + y)$ then $\gcd(x, x + y) = \gcd(x, y)$ too. Because $\gcd(1, 1) = 1$, all pairs of numbers reachable must also have gcd of 1, meaning they are coprime Q.E.D.

Also note that the number of ways to reach an ordered pair (x, y) is 1 because we always subtract the smaller number from the bigger number in the Euclidean algorithm, meaning there is only one way to go backward (the $x = y$ case only happens once, at $(1, 1)$ where there is no way to go back).

Now, think about a slow solution. We start at $(1, 1)$, and then loop through a queue. Every time we $ax + by <= c$, we add both $(x + y, y)$ and $(x, x + y)$ to the queue. The answer would be the number of elements in the queue where $ax + by > c$. But another way to compute the answer is to start with 1 and add 1 every time we do the adding to the queue. This is because every time we add 2 new possibilities, we remove 1, so overall the number of possibilities is added by 1. So the problem simplifies to finding the number of coprime pairs such that $ax + by <= c$ and adding 1. Counting the number of coprime pairs under a line can be done by loop through all x's, computing the max y at an x, and using PIE on the prime numbers to count the number of coprime pairs with a fixed x and a y going from 1 to max y. To do the PIE step we essentially maintain all numbers that contain all distinct prime factors and take the sum of

$$\frac{\max y}{\text{number}} \cdot (-1)^{\text{number of prime factors}}$$

In other words we use the Mobius function.

12 Modulo Queries

Observation 1: $X \pmod Y$ is equivalent to $X - Y \cdot \text{floor}(X/Y)$.

Using the property above, let's simplify the expression. $(A_l \pmod x) + (A_{l+1} \pmod x) + ... + (A_r \pmod x) \Rightarrow (A_l + A_{l+1} + ... + A_r) - x * [\text{floor}(A_l/x) + \text{floor}(A_{l+1}/x) + ... + \text{floor}(A_r/x)]$

The first part $(A_l + A_{l+1} + ... + A_r)$ can be found using prefix sums, so we're left with finding $\text{floor}(A_l/x) + \text{floor}(A_{l+1}/x) + ... + \text{floor}(A_r/x)$.

Observation 2: $\text{floor}(A_i/x)$ for any A_i can only take on $O(\text{sqrt}(A_i))$ values.

Thus, for each i, we can create a list of x where $\text{floor}(A_i/x)$ changes (e.g. for $A_i = 10$, we would have $x \in 1, 2, 3, 4, 10$ which would result in $\text{floor}(A_i/x) = 10, 5, 3, 2, 1$ respectively). Each list will have size at most $O(\text{sqrt}(A_i))$.

Then, sweep for increasing values of x and maintain a data-structure that can handle point updates and range sum queries.

There are a total of $O(N\text{sqrt}(A_i))$ updates and $O(Q)$ queries on the data-structure, so using a fenwick tree or a segment tree would result in $O(N\text{sqrt}(A_i) \log N + Q \log N)$ complexity.

We can do better! It's possible to optimize this using square root decomposition: Split the array into blocks of size $\text{sqrt}(N)$ and maintain the sum of each block. A range query can be split into the sum of entire $O(\text{sqrt}(N))$ blocks

and $O(\text{sqrt}(N))$ individual elements. Updates are $O(1)$ and queries are $O(\text{sqrt}(N))$, so $O(N\text{sqrt}(A_i))$ updates $+O(Q)$ queries $\Rightarrow O(N\text{sqrt}(A_i) + Q\text{sqrt}(N))$ time complexity.

The current memory complexity is $O(N\text{sqrt}(A_i))$ if the list of updates is precomputed. This can further be optimized to $O(N)$ memory by computing the next changing point of $\text{floor}(A_i/x)$ after updating the current changing point (if the current changing point is x the next changing point will be $\text{floor}(A_i/\text{floor}(A_i/x)) + 1$).

In total, the time complexity is $O(N\text{sqrt}(A_i) + Q\text{sqrt}(N))$ and the memory complexity is $O(N)$.