Battle of Brains 2022 Replay Analysis

Jubayer Nirjhor

October 2023

Contest Link. https://codeforces.com/gym/104679

Contents

A First Year, Second Year

Setter. Md Ibrahim Khandakar **Tester.** Zaiyan Ahnaf Rahim

Abridged Statement. Given the sum and the difference of two positive integers, recover them.

Solution. Given $S = x + y$ and $D = x - y$, one can add the equations to find $x = \dfrac{S + D}{2}$, and subtract the equations to find $y = \dfrac{S - D}{2}$.

B Even Out

Setter. Asif Jawad **Tester.** Prothito Shovon Majumder

Abridged Statement. Given a list of integers, you can flip the sign of exactly one of these numbers. In how many ways can you do this so that the sum of the resulting numbers is even?

Solution 1. Since the constraints are rather low, you could actually simulate the entire process. You can iterate over each i, flip the sign of A_i, find the sum of all the numbers, then check if the sum is even. This results in an $\mathcal{O}(n^2)$ solution.

Solution 2. You can find the initial sum and call it S. Notice that flipping the sign of a number A_i means the sum of the whole array changes to $S - 2A_i$. You can simply check if this value is even for each i and count the number of valid solutions. This results in a faster $\mathcal{O}(n)$ solution.

Solution 3. This solution is more clever. You can notice that the sum of a bunch of numbers is even if and only if the count of odd numbers is even. But flipping the sign keeps an odd number odd, and an even number even. So flipping the sign actually does not matter at all.

- If the sum of all the numbers is even at the beginning, then you can flip any of them and the resulting sum will still be even. So the answer is n in this case.
- If the sum is odd at the beginning, then there's no way to make it even by flipping signs. So the answer is 0 in this case.

C Odd One Out

Setter. Prothito Shovon Majumder **Tester.** Zaiyan Ahnaf Rahim

Abridged Statement. For a positive integer X, the number of ordered pairs (a, b) of positive integers satisfying $\text{GCD}(a, b) \times \text{LCM}(a, b) = X$ is denoted by $f(X)$. Find the number of $X \in [L, R]$ for which $f(X)$ is odd.

Solution. Using the well-known identity $\text{GCD}(a, b) \times \text{LCM}(a, b) = ab$, one can see that $f(X)$ is simply the number of divisors of X. The only positive integers with an odd number of divisors are the perfect squares. So the problem really asks you to count the number of perfect squares within the range $[L, R]$. As there are exactly $\left\lfloor \sqrt{k} \right\rfloor$ perfect squares not exceeding k, the answer is just $\left\lfloor \sqrt{R} \right\rfloor - \left\lfloor \sqrt{L-1} \right\rfloor$.

There are several ways to compute this. You can apply binary search, or use a square root function provided by your favorite language. The constraints actually allow you to just run a loop as well.

D Yet Another Mysterious Array

Setter. Mohidul Haque Mridul **Tester.** Zaiyan Ahnaf Rahim

Abridged Statement. Two players play a game on an array of positive integers and alternate moves. In each move, the player has to choose a prime number p that divides at least one number in the array, and then divide every such number in the array by p (if it's divisible by p). The player who cannot make a move loses. Find the winner.

Solution. The first thing to notice is that different primes do not affect the moves of each other. Let's consider a fixed prime p. How many times can this prime be chosen for a move? Each such move reduces the exponent of p in the prime factorization of every number in the array by 1 (if it was nonzero before). As long as at least one number is divisible by p (has a nonzero exponent on p in its prime factorization), one can make a move using p. Then it's evident that the prime p can be chosen exactly t times, where t is the maximum exponent of p in any number in the array.

We can, hence, compute the total number of moves that the game will last. It's simply the sum of the number of moves possible for every prime p (which is the maximum exponent of p in the prime factorization of any number in the array). This computation can be aided by a precomputed table containing the smallest prime factor of each relevant number, which you can find via sieve. The constraints are low enough to even allow some factorization methods that exhaust numbers up to the square root.

Once we know the total number of moves that the game will last, we know who makes the last move (inspecting the parity of the number of moves). Since the last player to make a move wins, we deduce that the first player wins if and only if the total number of moves is odd.

E Rasta Thamaye Dilo

Setter. Rahat Hossain **Tester.** Prothito Shovon Majumder

Abridged Statement. There's a special graph on vertices numbered from 2 to n. In this graph, there is an edge between u and v if and only if one of them divides the other. What is the minimum number of new edges needed to make the graph connected?

Solution. Let's understand how things are connected to each other.

- Any number $x \le \frac{n}{2}$ is connected to 2 because x has an edge to $2x$ which has an edge to 2. Since $x \le \frac{n}{2}$ the number $2x \le n$ and it exists in our graph.
- Any composite number x has a factor $\le \frac{x}{2}$ (since the smallest prime factor is at least 2). So any composite number $x \le n$ is connected to a number $\le \frac{n}{2}$ which in turn is connected to 2 (based on the above observation).
- Any prime number $p > \frac{n}{2}$ is an isolated vertex (has no edge to any other vertex). This is because the only smaller factor of p is 1 which does not exist in our graph. And the next smallest multiple of p is $2p$ which is bigger than n due to $p > \frac{n}{2}$. So $2p$ does not exist in our graph either.

Based on these, we can deduce that every prime number p bigger than $\frac{n}{2}$ are isolated vertices. Everything else is already connected to 2. So one can simply connect those isolated prime numbered vertices to 2 and get the job done. The answer is, hence, simply the count of prime numbers in the range $\left(\frac{n}{2}, n\right]$. You can run a sieve in order to find all the relevant prime numbers beforehand. There are two possible approaches to answer multiple test cases fast.

- For each number i, you could store the value $\texttt{prefix}[i]$, which is the number of prime numbers up to i. Then the answer can be retrieved as $\texttt{prefix}[n] - \texttt{prefix}[\frac{n}{2}]$.
- You could store those prime numbers in an array (in sorted order), and then run binary search every time to find the number of primes in a specific range.

Finally, be careful of the cases $n = 2$ and $n = 3$. In these cases, the value of $\frac{n}{2}$ goes below 2, so the count of primes includes 2. This leads to a wrong result. You can simply handle these two cases manually.

F Lucky Seats

Setter. Wakilur Islam **Tester.** Mohidul Haque Mridul

Abridged Statement. For a set S of distinct non-negative integers, we are given two numbers: the bitwise or of these numbers, and the bitwise xor of these numbers. What is the maximum number of integers that can be in that set?

Solution. Make sure you understand how these bitwise operations work on a set of numbers. Let's call the value of bitwise or O, and the value of bitwise xor X. We consider their binary representations (after all, we care about bitwise operations here). If some bit is 0 in O, then none of the numbers can have a 1 on that position. This means there should also be a 0 in X on that position. This lets us get rid of some special cases.

- If X is not a subset of O, then there is no solution (so the answer is -1). Here, a number x is a subset of another number y if there is no bit position where x has a 1 while y has a 0. You can verify that x is a subset of y if and only if $x \wedge y = x$, where $\wedge$ is bitwise and operation (think why).
- If $O = 0$, then the only possible number in the set is 0. In this case we must also have $X = 0$, and we can print the answer.

Now let's say there are $k > 0$ bits that are 1 in O, and we call these bit positions *allowed positions*. So we already have an upper bound on the maximum size of S — it is 2^k (all possible numbers with 1 bits in a subset of the allowed positions). We get rid of one more special case.

- If there's exactly $k = 1$ allowed position, then there are only two possible values that can be in the set. They are 0 and O. We cannot only have 0 in the set as then $O = 0$. So we must have both 0 and O in the set. In this case $X = O$ and we can print the answer accordingly.

Now we deal with the case of $k \geq 2$ allowed positions. Consider the set of all 2^k numbers with 1 bit in a subset of these positions. There are exactly 2^{k-1} numbers with 1 bit in a specific allowed position. Since 2^{k-1} is even for $k \geq 2$, the bitwise xor of all these numbers is 0. Therefore, if $X = 0$ then we can take all of these numbers.

What if $X \neq 0$? Remember that X has to be a subset of O for a solution to exist. So the number X exists in those 2^k numbers that we're considering. If we remove X from these numbers, then the bitwise xor of the remaining $2^k - 1$ numbers becomes X (think why this is true, using the fact that the xor was 0 before). So the answer is simply the set of these $2^k - 1$ numbers.

As an example, consider $O = 5 = \texttt{101}_2$ with $k = 2$ and $X = 4 = \texttt{100}_2$. We consider the $2^k = 4$ numbers $\{\texttt{000}_2, \texttt{001}_2, \texttt{100}_2, \texttt{101}_2\}$ (subsets of allowed positions). These numbers have bitwise or 5 but bitwise xor 0. If we remove $X = \texttt{100}_2$ from this set, we're left with $\{\texttt{000}_2, \texttt{001}_2, \texttt{101}_2\}$ with the desired values of bitwise or $O = 5$ and bitwise xor $X = 4$.

G Winter Gifts

Setter. Ayon Shahrier **Tester.** Mohidul Haque Mridul

Abridged Statement. You can perform the following operation on a string: choose two indices i, j from the string satisfying $|i - j| = k$ and change the i-th character into the j-th character. You're given the value of k. Can you transform the given string S (performing the stated operation some number of times) into the given string T.

Solution. Consider the strings to be 0-indexed. Notice that index 0 can affect index k, which in turn can affect index $2k$ and so on. Similarly, index 1 can affect index $k + 1$, which in turn can affect index $2k + 1$ and so on. Essentially, we have k groups of indices (one starting with each of $\{0, 1, \dots, k - 1\}$). These groups are $\{0, k, 2k, \dots\}, \{1, k + 1, 2k + 1, \dots\}, \dots, \{k - 1, 2k - 1, 3k - 1, \dots\}$. The mentioned operation can only affect indices in a particular group, and different groups are not affected simultaneously by a single operation. So we can extract k strings out of S and T — one for each of the groups of indices. As an example, if $S =$ `abcdefgh` and $k = 3$ then the 3 strings are $\{$`adg`, `beh`, `cf`$\}$. Then we have to answer k questions of the following form:

- You can perform the following operation on a string: pick a position and change it to one of its adjacent characters. Can you transform the given string S to the given string T?

In short, we've reduced the problem to the $k = 1$ case. Consider a compression of string T where we replace a contiguous block of same character with a single character. For example, we compress $T =$ `bbbhhpgggaa` into $T_{\text{short}} =$ `bhpga`.

Claim. The answer is yes if and only if T_{short} is a subsequence of S.

Proof. Let's show the necessity first. Due to the nature of the operation, we cannot swap two adjacent characters, we can only replace one by the other. Also, we cannot place new characters. So if T_{short} is not a subsequence of S at the beginning, it will never be a subsequence of S after any number of operations (since there's never any swap of adjacent characters). But T_{short} is a subsequence of T, so we cannot transform S into T if T_{short} is not a subsequence of S in the beginning.

Now let's show the sufficiency. If T_{short} is a subsequence of S, then we can move T_{short} to the beginning of S. This can be done by going through characters of T_{short} from left to right, finding its corresponding position in S (in the subsequence), and dragging it to the front (in the correct position). Then we can build T by going through characters of T_{short} from right to left, finding its corresponding position in the front of S (remember we moved T_{short} to the front of S), dragging it to the right in the suitable position suggested by T. As we make many copies of the character when dragging it, we'll have exactly the string T after this algorithm. $\square$

This gives us a way to answer the modified question in linear time, solving the whole problem in linear time as well.

H A Dance with DS

Setter. Asif Jawad **Tester.** Mohidul Haque Mridul

Abridged Statement. Given a positive integer k and a limit r. For a positive integer n, if it's divisible by k then you divide it by k, and otherwise you subtract 1 from it. Let $f(n)$ denote the number of steps to get to 0 from n. Find the maximum of $f(n)$ over all $0 \le n \le r$.

Solution. Let's analyze the case of $n = 274$ and $k = 10$.

- Subtract one 4 times until it gets to $n = 270$. Divide it by $k = 10$ and get $n = 27$.
- Subtract one 7 times until it gets to $n = 20$. Divide it by $k = 10$ and get $n = 2$.
- Subtract one 2 times until it gets to $n = 0$.
- We needed $f(274) = (4 + 1) + (7 + 1) + 2 = 15$ operations.

If you notice carefully, in the case of $k = 10$ we essentially reduce the last digit to 0 and then remove it ($274 \to 270 \to 27 \to 20 \to 2 \to 0$). So we can deduce that

$$f(n) = (\text{sum of digits of } n) + (\text{number of digits of } n) - 1.$$

In fact, if you notice why this is true for $k = 10$, you can figure out that this formula holds for any value of k — we just have to represent n in base k. As an example, for $k = 2$ we have $f(12) = 5$ because the path is $12 \to 6 \to 3 \to 2 \to 1 \to 0$. But in binary (base $k = 2$), this path is $\texttt{1100}_2 \to \texttt{110}_2 \to \texttt{11}_2 \to \texttt{10}_2 \to \texttt{1}_2 \to \texttt{0}_2$. The same formula holds here.

So we need to find the maximum of (sum of digits of n) + (number of digits of n), in base k, over all numbers n below r. The idea is that any number n less than r would have a certain number of digits (maybe zero) from the left in common with r, and then a digit which is smaller than the corresponding digit in r. We can iterate over the size of that common portion (in base k of course). After that, for the following digit we can take one less than the corresponding digit in r, as we want to maximize the sum of digits (notice that we cannot do this if that following digit is 0). As the number becomes smaller than r after that, we can assign the rest of the digits to be the maximum, which is $k - 1$.

As an example, consider $k = 9$ and $r = 41308_9$. One possible candidate is fixing the length of the common portion as 2. Then under that constraint, the optimal choice is the number $n = 41288_9$. Here we reduced the third digit (from left) of r by 1 and replaced all the following digits with $k - 1 = 8$. We take the maximum over all such candidates.

Be careful about the case of zero common prefix with r, in which case there's a possibility of the number of digits reducing by 1 (if the leading digit of r is 1 in base k).

I Stairway To Heaven

Setter. Mohidul Haque Mridul **Tester.** Sakib Hassan

Abridged Statement. Consider a directed graph on the positive integers. There's an edge from u to v if and only if $u < v$ and $\mathrm{GCD}(u, v) = 1$. Find the number of paths from S to T modulo a large prime.

Solution. Let $\texttt{count}[x]$ denote the number of paths from x to T. Clearly we have

$$\texttt{count}[x] = \sum_{\substack{x<y\le T \\ \mathrm{GCD}(x,y)=1}} \texttt{count}[y]$$

with $\texttt{count}[T] = 1$. Computing the values of $\texttt{count}[x]$ in descending order of $x \in [S, T]$ gives us an $\mathcal{O}(T^2)$ solution, our desired answer being $\texttt{count}[S]$.

To speed this up, let's define

$$\texttt{sum}[x] = \sum_{k=1}^{\left\lfloor \frac{T}{x} \right\rfloor} \texttt{count}[kx]$$

to be the sum of $\texttt{count}[y]$ over all multiples y of x. How does that help us compute $\texttt{count}[x]$? Let's consider the set P of distinct prime factors of x. The numbers y with $\mathrm{GCD}(x, y) \ne 1$ are the ones divisible by at least one of the primes in P. We can apply the principle of inclusion-exclusion to come up with

$$\texttt{count}[x] = \sum_{P_{\text{subset}} \subseteq P} (-1)^{|P_{\text{subset}}|} \texttt{sum}\left[\prod P_{\text{subset}}\right]$$

where $\prod P_{\text{subset}}$ denotes the product of the elements in P_{subset}. Intuitively, we add $\texttt{count}[y]$ for all y, then for each prime in P we remove $\texttt{count}[y]$ for the multiples y of that prime, but then we need to add back $\texttt{count}[y]$ for the product of two different primes which we removed twice, and so on.

Since we're only interested in the values of $\texttt{sum}[y]$ for square-free numbers y, once we compute $\texttt{count}[x]$ for some number x, we can update all the relevant values of $\texttt{sum}[y]$ via the same procedure. We simply iterate over $P_{\text{subset}} \subseteq P$ and add $\texttt{count}[x]$ to $\texttt{sum}[\prod P_{\text{subset}}]$. It is possible to precompute the sets P for each number till T in a sieve-like manner. This allows us to compute the values of $\texttt{count}[x]$ and $\texttt{sum}[x]$ in descending order of $x \in [S, T]$, having to iterate only on the square-free divisors of x. The time complexity is bounded by $\mathcal{O}(T \lg T)$ while requiring $\mathcal{O}(T)$ space.

J XORted

Setter. Sakib Hassan **Testers.** Nayeemul Islam Swad, Jubayer Nirjhor

Abridged Statement. There's a sorted array A of size n and a lot of queries. Each query gives you a range $[L, R]$ of the array. You can choose a non-negative integer $X < 2^{20}$ and assign $A[i] := A[i] \oplus X$ for all $L \le i \le R$, where $\oplus$ denotes bitwise xor. How many values of X can you pick such that the whole array remains sorted after the operation?

Solution. The idea is to notice that to keep the array sorted, we need to ensure $A[i] \le A[i+1]$ for each $1 \le i < n$.

Let's try to answer the following question. If there are two numbers $P \le Q$, then which values of X ensure that $P \oplus X \le Q \oplus X$? We naturally consider the binary representations of the numbers. Notice that if $P \le Q$ then there is a common prefix (portion from the left) between P and Q (possibly all of it), and the following digit has a 0 in P and a 1 in Q (this is where Q becomes larger than P). For example, $P = 9 = \texttt{1001}_2$ and $Q = 10 = \texttt{1010}_2$ satisfies $P \le Q$. They have a common prefix until the second digit (from left), followed by the third digit where P has a 0 and Q has a 1. When we xor both of these numbers by a fixed number X, the bits where X has a 1 get flipped in both P and Q, with the rest of the bit positions remaining unaffected. That means the length of the common prefix does not get affected at all. So we cannot flip the bit at the position following the common prefix (in the example, we cannot flip the third bit, so X cannot have a 1 in that position) because in that case P would become bigger than Q. If we can ensure that, the rest of the bits in X can be anything. In conclusion, at most one bit position is *banned* from having a 1 in the binary representation of X (no restriction if $P = Q$).

Back to our problem, consider a position i with $L \le i < R$. Then we would xor both $A[i]$ and $A[i+1]$ with the number X. If we want to keep the array sorted, then according to the above analysis, at most one bit position of X gets banned from having a 1 in that position. We can precompute these positions for each position $1 \le i \le n$ in the array, and then extract the banned positions in range $L \le i < R$ for a query (there are at most $\lg \max A = 20$ bit positions, so it can be done fast). So X can't have 1 in those bit positions. But what about the other bit positions? Can X have anything on those bits?

The problem is with the endpoints L and R. To keep the whole array sorted, we should choose X such that both $A[L-1] \le A[L] \oplus X$ and $A[R] \oplus X \le A[R+1]$ hold (for ease of implementation, we can assign $A[0] = 0$ and $A[n+1] = 2^{20} - 1$). These inequalities pose some additional restrictions on X, which can be met via dynamic programming on the digits of X.

We can assign the digits of X from left to right. Only three pieces of information are really necessary here as we go forward.

- Which digit from the left of X are we currently assigning? This value can range from 0 to $\lg \max A = 20$. We'll assign the digits from the left to right (decreasing order of the digit index).

- Is the current value of $A[L] \oplus X$ (with the digits assigned so far on X) strictly bigger than $A[L-1]$? This can simply be encoded with 0 and 1 denoting yes and no, respectively. If the answer is no, that means we're still on the common prefix between $A[L-1]$ and $A[L] \oplus X$, so we cannot choose a digit on X that will make the corresponding digit of $A[L] \oplus X$ smaller than that digit of $A[L-1]$. If the answer is no, then we've already made $A[L] \oplus X$ strictly bigger than $A[L-1]$, so further digits of X get no restriction from this side.

- Very similarly, is the current value of $A[R] \oplus X$ (with the digits assigned so far on X) strictly smaller than $A[R+1]$? This can also be encoded with 0 and 1 denoting yes and no, respectively. If the answer is no, that means we're still on the common prefix between $A[R+1]$ and $A[R] \oplus X$, so we cannot choose a digit on X that will make the corresponding digit of $A[R] \oplus X$ bigger than that digit of $A[R+1]$. If the answer is no, then we've already made $A[R] \oplus X$ strictly smaller than $A[R+1]$, so further digits of X get no restriction from this side.

These pieces of information span a space of size $2 \times 2 \times \lg \max A \approx 80$. So we can run a dynamic programming solution for every query. Keep in mind that we possibly have a bunch of banned positions from the indices $L \leq i < R$ where we cannot put a 1, so in our recursive function we have to check for that as well. Overall, this solution runs in $\mathcal{O}((n+q) \lg \max A)$ time and memory where q is the number of queries.