Competitive Programming Problem Editorials

2043A - Coin Transformation

Let's try to solve this problem *naively*: obviously, while we have at least one coin with value > 3, we should transform it since it increases the number of coins we get. We can simulate this process, but the number of transformations might be really large, so we need to speed this up.

Let's make it faster in the following way: instead of transforming just one coin, we will transform all coins at once. After one operation, we will have 2 coins with value $\frac{n}{2}$; after two operations, we will have 4 coins with value $\frac{n}{4}$ each, and so on. This can be implemented using a simple `while` loop: while the value of our coins is greater than 3, we divide it by 4 and double the number of coins. This solution works in $O(\log n)$.

It is also possible to derive a formula for the answer: the number of times we need to divide a number by 4 so it becomes less than 4 is $\lfloor \log_4 n \rfloor$, and the number of coins we will get is $2^{\lfloor \log_4 n \rfloor}$. However, you must be very careful with this approach because it can have severe precision issues due to the fact that standard logarithm functions work with floating-point numbers, which are imprecise. Instead, use a method to calculate $\log_a n$ without floating-point calculations, for example, iterating (or binary searching) on the power of 4 you need to divide the number by to make it less than 4.

2043B - Digits

There are several ways to solve this problem. I will describe two of them.

Using Divisibility Rules (A Lot of Math Involved)

We can try divisibility rules for all odd integers from 1 to 9 and find out whether they work for our numbers:

- 1 is always the answer since every integer is divisible by 1.
- A number is divisible by 3 if its sum of digits is divisible by 3. Since our number consists of $n!$ digits d, either $n!$ or d should be divisible by 3; so, $n > 3$ or $d \bmod 3 = 0$.
- A number is divisible by 9 if its sum of digits is divisible by 9. This is a bit trickier than the case with 3 because it is possible that both $n!$ and d are divisible by 3 (but not 9), making the sum of digits divisible by 9.
- A number is divisible by 5 if its last digit is 5 or 0. Just check that $d = 5$, and that's it.

- The trickiest case: A number is divisible by 7 if, when split into blocks of 3 digits (possibly with the first block shorter than 3 digits), the sign-alternating sum of these blocks is divisible by 7. For example, 1234569 is divisible by 7 because $(1-234+569)$ is divisible by 7. If $n > 3$, the number can be split into several blocks of length 6, and each such block changes the alternating sum by 0. Thus, if $n > 3$ or $d = 7$, the number is divisible by 7.

Almost Brute Force (Much Less Math Involved)

First, we need a little bit of math. If you take a number consisting of $n!$ digits equal to d, it is always divisible by $(n-1)!$ digits equal to d. This is because, if you write some integer repeatedly, the resulting number will be divisible by the original number. For example, 424242 is divisible by 42.

If for some $n = k$, the number is divisible by some digit, then for $n = k + 1$, the number will also be divisible for the same digit.

This means there exists an integer m such that for all integers $n > m$, the results are the same if you use the same digit d. Thus, we can set $n = \min(n, m)$, and if m is small enough, use brute force.

The value of m? The samples tell us that the number consisting of 7! ones is divisible by 1, 3, 7, and 9 (divisibility by 5 depends only on d), so you can use $m = 7$. It is possible to reduce m to 6, but this is not required.

The solution is to reduce n to something like 7 if it is greater than 7, then use brute force. You can either calculate the remainder of a big number modulo a small number using a `for`-loop, or, if coding in Java or Python, use built-in big integers (just be careful with Python—modern versions of it forbid operations with integers longer than 4300 digits; you might need to override that behavior).

2043C - Sums on Segments

What could the answer to the problem be if all elements were equal to 1 or -1? Let's consider all segments with a fixed left boundary L. The empty segment has a sum of 0. As we move the right boundary to the right, the sum will change by ± 1. That is, we can obtain all sums from the minimum sum to the maximum one. To find the sums for the entire array, we need to find the union of the segments. Since all segments include 0, their union is also a segment. Therefore, the possible sums are all sums from the minimum sum to the maximum sum in the entire array.

Now let's apply this reasoning to the given problem. The segments that do not include the "strange element" still form a segment of possible sums that includes 0. As for the segments that include the strange element, we can look at them this way: we will remember these segments and remove the strange element. Then the resulting sums will also form a segment that includes 0. If we return the element to its place, all sums will increase exactly by this element. Thus, it will remain a segment, though not necessarily including 0 now.

The solution is as follows:

1. Find the minimum and maximum sum among the segments that do not contain the strange element.

2. Find the minimum and maximum sum among the segments that do contain it.
3. Output the union of the obtained sum segments.

Next, adapt your favorite algorithm for finding the maximum sum segment for this problem. My favorite is reducing it to prefix sums. The sum of the segment $[i, r]$ is equal to $\text{pref}[r] - \text{pref}[i-1]$. We fix the right boundary of the segment r. Since the first term for all segments is now the same, the maximum sum segment with this right boundary is the one with the minimum possible prefix sum at the left boundary. We then iterate over r in increasing order and find the maximum sum among all right boundaries. The minimum prefix sum on the left can be maintained on the fly.

For a fixed right boundary, there are two options:

- For some prefix of left boundaries, the strange element is inside the segment.
- For some suffix, it is outside (this suffix may be empty if the boundary r is to the left of the strange element).

Maintain two values on the fly: the minimum prefix sum before the strange element and after it.

Finally, find the possible sums in the union of the two segments:

- If the segments intersect, then it includes all sums from the minimum of the left boundaries to the maximum of the right ones.
- If they do not intersect, it is simply two segments.

Overall complexity: $O(n)$ per test case.

2043D - Problem About GCD

First, let's solve the problem with $G = 1$. We can check the pair (l, r). If its greatest common divisor is not 1, then we check $(l, r-1)$ and $(l+1, r)$, i.e., the pairs at a distance $(r - l - 1)$. If these don't work, we check $(l, r-2)$, $(l+1, r-1)$, and $(l+2, r)$, and so on. The answer will be located fast enough (as explained in the third paragraph of the editorial). This gives a solution in $O(K^2 \log A)$ per test case, where A is the bound on the integers in the input, and K is the decrease in distance required to find the pair (l, r) (i.e., if the answer has distance $|A - B|$, then $K = r - l - |A - B|$).

What to do if $G > 1$? The approach is almost the same, but first, ensure that the first pair we try has both integers divisible by G. If $l \mod G \neq 0$, shift l to the next closest integer divisible by G. Similarly, if $r \mod G \neq 0$, subtract $r \mod G$ from r to make it divisible. Then, proceed as before, but instead of trying pairs like $(l, r-1)$, $(l+1, r)$, $(l, r-2)$, and so on, try $(l, r-G)$, $(l+G, r)$, $(l, r-2G)$, and so on.

Now let's talk about why this works fast (i.e., why K in the complexity formula is not large). All following paragraphs assume $G = 1$, but the same reasoning applies for $G > 1$ if we divide everything by G.

Intuitively, we can think about it in terms of prime gaps: as soon as $r - K$ becomes a prime number, we get our result. The average gap between two primes is about $\ln A$, but there can be relatively large gaps, sometimes exceeding 1000. If you're bold and brave, you can stop here and submit, but let's find a better bound.

Instead of thinking about the gap between two primes, consider the gap between two numbers that are coprime with l. Assume l is the product of several of the first prime numbers. If it is not, integers coprime with l will appear even more frequently. For example:

- $\frac{1}{2}$ of all integers are not divisible by 2.
- $\frac{2}{3}$ of the remaining integers are not divisible by 3.
- $\frac{4}{5}$ of the remaining integers are not divisible by 5.

Repeating this process until the product of primes becomes too large, we find that, on average, 1 in 7 or 8 integers is coprime with l.

This is a better bound, though it still uses "average" gaps. However, it should be sufficient to attempt a submission.

For a rigorous proof, consider all possible pairs of integers from intervals $[l, l+30)$ and $[r-30, r]$. There are 30 integers in each interval, so 900 pairs to consider. Let us analyze:

- If both integers are divisible by 2, there will be at most 225 such pairs, leaving 675 pairs.
- If both integers are divisible by 3, there will be at most 100 such pairs, leaving 575 pairs.
- If both integers are divisible by 5, there will be at most 36 such pairs, leaving 539 pairs.

Continuing this until a prime like 37, we find at least one coprime pair exists for all integers $\leq 10^8$ in intervals of size 30. Thus, $K < 60$. In practice, K is much smaller, as this proof doesn't account for numbers divisible by multiple primes.

2043E - Matrix Transformation

Every operation that affects multiple bits can be split into several operations that affect one bit. For example, performing an OR operation with $z = 11$ is equivalent to performing three OR operations with $z = 1$, $z = 2$, and $z = 8$. Let's solve the problem for each bit separately.

Considering one bit, the problem becomes:

- Given a binary matrix A, determine if it can be transformed into another binary matrix B.
- The allowed operations are "set all elements in a row to 0" or "set all elements in a column to 1."

To solve:

1. Find all cells where $A[i][j] \neq B[i][j]$.
2. If $A[i][j] = 0$ and $B[i][j] = 1$, apply an operation to the j-th column.

3. If $A[i][j] = 1$ and $B[i][j] = 0$, apply an operation to the i-th row.

Not all operations are independent. Suppose $B[i][j] = 0$ and we change the j-th column to 1. Then, we must apply an operation to the i-th row to reset $B[i][j]$ to 0. Similarly, for $B[i][j] = 1$, an operation on the i-th row necessitates one on the j-th column.

To handle dependencies:

- Build a directed graph where vertices represent operations and an edge $z \to y$ implies z must precede y.
- If a cycle exists in the graph, the transformation is impossible.
- Otherwise, perform a topological sort and apply operations in that order.

Cycle detection can be achieved using a "three-colored DFS":

- A vertex is white if unvisited, gray if on the DFS stack, and black if fully processed.
- A back-edge to a gray vertex indicates a cycle.

If transformations for all bits are possible, output "Yes"; otherwise, output "No." Complexity: $O(nm \log A)$.

2043F - Nim

The second player wins in "Nim" if the XOR of the pile sizes is 0. The task is to remove as many piles as possible such that the XOR becomes 0.

Key observation:

- Suppose there are c piles of size z in a segment. Removing an even number of piles leaves the XOR unchanged, while removing an odd number changes it by z.
- Thus, there is no benefit in keeping more than 2 piles of the same size. If $t > 2$ piles exist, remove 2 without affecting the outcome.

To answer queries:

1. Precompute the count of each element in each prefix.
2. Use dynamic programming where $dp[i][j][f]$ represents:
   - Maximum number of removed elements.
   - Number of ways to achieve this.
   - $f = 1$ if at least one element remains; otherwise, $f = 0$.

Transitions:

- Remove all elements of a value: 1 way.
- Keep 1 element: c ways.
- Keep 2 elements: $\binom{c}{2}$ ways.

Base case: $dp[0][0][0] = (0, 1)$; others initialized to $(-1, -1)$. Final state: $dp[51][0][1]$. If $(-1, -1)$, no solution exists.

Complexity: $O(51 \times 64 \times 2)$ states with 3 transitions each, fitting within constraints.