Problem A. Find the metro station parent

We use proof by recursion to determine the structure of the children.

$$\text{FirstChild}(N) = \sum_{i=1}^{N-1} i + 2 = \frac{(N-1)N}{2} + 2$$

By definition, the last child of N is:

$$\text{LastChild}(N) = \text{FirstChild}(N) + N - 1$$

Thus, the children of N lie in the range:

$$[\text{FirstChild}(N), \text{LastChild}(N)]$$

To find the parent of X, perform a linear search:

- Start with $N = 1$.
- Compute $\text{FirstChild}(N)$ and $\text{LastChild}(N)$.
- If X is in the range, return N.
- Otherwise, increment N and repeat.

Each test case runs in $O(\sqrt{N})$ time complexity since each parent has children in the range $O(N^2)$. The memory complexity is $O(1)$ since we do not need to store any variable amount of data.

Problem B. Affordable Rent

To efficiently handle queries on properties in a 2D space, we need a data structure that supports fast point insertions and rectangular range queries. The two main choices are:

- **2D Segment Tree**: A segment tree where each node represents a square region instead of a 1D range, allowing efficient queries over 2D spaces.
- **Quadtree**: A space-partitioning tree that recursively subdivides the 2D plane into four quadrants, dynamically adapting to point distributions.

Both structures have the same theoretical time complexity:

- **Insertion:** $O(\log N + \log M)$
- **Rectangle query:** $O(\log N \cdot \log M)$

where N and M represent the dimensions of the space. However, the quadtree offers practical advantages:

- **Reduced Heap Allocations:** The 2D segment tree requires frequent dynamic memory allocations, whereas the quadtree minimizes heap allocations which are very slow, leading to better performance.
- **Better Cache Locality:** The quadtree keeps spatially close points together, reducing cache misses and improving performance.

While both approaches are valid, the quadtree often provides better performance due to its reduced memory overhead and improved cache efficiency.

Problem C. Minimum Operations

1. Identify the first occurrence of ‘1‘ in S.
2. Handle special cases:

- If S has no ‘1‘s, it requires N operations (one increment and $N-1$ shifts).
- If the first ‘1‘ is already at the leftmost position but other ‘1‘s exist, return -1 (impossible transformation).
- If there is exactly one ‘1‘, shift it left until it reaches the first position.

3. Otherwise, count the increments needed to fill all ‘0‘s between the first ‘1‘ and the end:

$$\text{fillZeros} = \sum_{bit=0}^{firstOne} (1 - S[bit]) \cdot 2^{bit} \mod (10^9 + 7)$$

4. Perform increments before shifting:

Since left shifts multiply the numerical value of S by 2, shifting before filling the zeros makes increments exponentially more expensive. Thus, we first convert all necessary ‘0‘s into ‘1‘s, ensuring minimal increment cost, and then shift until $S = T$.

5. The total operations required:

$$\text{operations} = \text{fillZeros} + 1 + \text{shifts}$$

Problem D. Simple Game

Since each move reduces either m or n (or both), the game ends when the king reaches $(1, n)$ or (m, n). This is a classic combinatorial game theory problem, where the outcome depends on the parity of (m, n).

This is derived by analyzing the Grundy number ($Nimbers$) of the position, but it can be reduced to a simple parity check:

- If both m and n are odd, "Zakariaa"wins.
- Otherwise, "Anas"wins.

Each test case runs in $O(1)$ since it only requires checking the parity of m and n.

Problem E. Defending Xyron

This problem can be efficiently solved using the **Convex Hull Trick**, which optimizes queries over a set of linear functions. The data structure maintains a dynamic set of lines and efficiently finds the one that maximizes (or minimizes) a given query.

- **Insertion:** Amortized $O(1)$.
- **Query:** $O(\log N)$ using binary search.
- **Space Complexity:** $O(N)$ for storing the lines.

Another approach is the **Li Chao Tree**, which provides an alternative way to handle dynamic line insertion and queries in $O(\log N)$. Unlike the Convex Hull Trick, the Li Chao Tree is more flexible for handling arbitrary insertions but has slightly higher constant factors.

Problem F. Matrix Rank

You can use a trie, after finding the GCD of each row/col, we try to insert the elements of it to a trie after dividing them by the GCD, if the row already exists, then its lineary dependant to another row.

Another alternative is to use Hashing (Double hashing) to figure out if a two rows/columns are Lineary dependant.

Overall complexity $O(N.M)$ and Space $O(N.M)$.

Problem G. A Multiplication Problem

As explained, compute $a * b$ and be careful of possible overflows as variables are large

Problem H. Numerical Journey

Key Observation

Two numbers can be connected if they share at least one common prime factor. Instead of treating numbers directly, we invert the problem:

- Each number is represented by its **prime factors**.
- If two numbers share a common prime factor, they belong to the same connected component.
- If all numbers are part of the same connected component, then the answer is `"yes"`, otherwise `"no"`.

Solution Approach

1. Precompute the smallest prime divisor for every number up to 10^6 using the **Sieve of Eratosthenes**.
2. Use a **Union-Find** (DSU) structure where each **prime factor** is treated as a node.
3. For each number, decompose it into its prime factors and connect all the primes using Union-Find.
4. Finally, check if all prime numbers appearing in the factorization of any input number belong to the same connected component.

Time Complexity Analysis

- **Prime Factorization:** Each number has at most $O(\log n)$ prime factors.
- **Union-Find Operations:** Each union and find operation runs in nearly $O(1)$ with path compression.
- **Sieve of Eratosthenes Preprocessing:** Runs in $O(n \log \log n)$.

Thus, the overall complexity is: $O(n \log n)$

Final Check

Once Union-Find processing is done, we determine if all numbers are connected by checking whether all encountered prime factors belong to the same connected component.

Conclusion: If there is only one connected component, the answer is `"yes"`, otherwise, it is `"no"`.